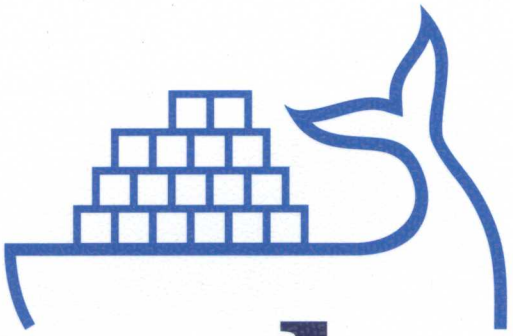
HZ BOOKS
华章IT

容器技术系列



Docker源码分析

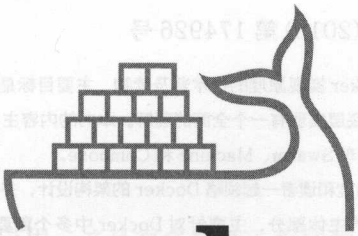
THE SOURCE CODE ANALYSIS OF DOCKER

孙宏亮 著

- 国内首部Docker源码分析著作
- 从源码角度全面解析Docker设计与实现
- 填补Docker理论与实践之间的鸿沟

机械工业出版社
China Machine Press

孙宏亮 硕士，浙江大学毕业，现为DaoCloud软件工程师，主要负责企业级容器云平台的研发工作。数年来一直从事云计算、PaaS领域的研究与实践，是国内较早一批接触Docker的先行者，同时也是Docker技术的推广者。



Docker 源码分析

THE SOURCE CODE ANALYSIS OF DOCKER

孙宏亮 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Docker 源码分析 / 孙宏亮著 . —北京：机械工业出版社，2015.8（2015.11 重印）
（容器技术系列）

ISBN 978-7-111-51072-7

Ⅰ. D… Ⅱ. 孙… Ⅲ. Linux 操作系统 – 程序设计 Ⅳ. TP316.89

中国版本图书馆 CIP 数据核字（2015）第 174926 号

本书是一本引导读者深入了解 Docker 实现原理的技术普及读物，主要目标是通过对 Docker 架构和源代码的详细讲解和解剖，帮助读者对 Docker 的底层实现有一个全面的理解。本书的内容主要集中于三部分：Docker 的架构、Docker 的模块，以及 Docker 的三驾马车 Swarm、Machine 和 Compose。

第一部分（第 1 章）主要从宏观的角度和读者一起领略 Docker 的架构设计，并初步介绍架构中各模块的职责。

第二部分（第 2 ~ 14 章）是本书的主体部分，主要针对 Docker 中多个重要的模块进行具体深入分析，包括 DockerClient、DockerDaemon、DockerServer、Docker 网络、Docker 镜像、Docker 容器等。读者可以发现，Docker 的模块之间耦合度非常低，很适合循序渐进，层层深入。第 2 ~ 8 章主要从 Docker 软件的架构入手，勾勒骨架；第 9 ~ 11 章集中于 Docker 镜像技术，夯实基础；第 12 ~ 14 章则进一步分析 Docker 容器的始末，阐述本质。

第三部分（第 15 ~ 17 章）介绍 Docker 生态三驾马车——Swarm、Machine 和 Compose。Docker 拥有强大的单机能力，三驾马车可以很好地补充 Docker 的跨主机能力以及部署能力。读者可以通过这几章感受 Docker 生态中其他功能强大的软件。

Docker 源码分析

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：陈佳媛 谢晓芳 责任校对：董纪丽
印 刷：北京市荣盛彩色印刷有限公司 版 次：2015 年 11 月第 1 版第 4 次印刷
开 本：186mm × 240mm 1/16 印 张：16.5
书 号：ISBN 978-7-111-51072-7 定 价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

Praise 赞 誉

像谷歌一样部署自己的应用，这是很多软件工程师的梦想。Docker 的目标是圆很多人的梦。自从 InfoQ 推出 Docker 系列文章，作为操作系统课程教师的我一直在学习并关注 Docker 的茁壮成长。

当我发现这上面刊登的“Docker 源码分析”系列文章的作者居然是我们课程组的研究生助教孙宏亮时，惊喜之情溢于言表。宏亮对 Docker 的理解十分深刻，他本人是 Docker 的积极拥护者、倡导者和贡献者。他在研究生毕业以后投身到了创业公司 DaoCloud，去为 Docker 的梦想开创美好的未来。

最近，我又欣喜地发现，这系列文章以及后续章节即将正式出版成书，有机会同更多的 Docker 用户、开发者、学习者见面。本书通过分析解读 Docker 源码，让读者了解 Docker 的内部结构和实现，以便更好地使用 Docker。该书的内容组织深入浅出，表述准确到位，有大量流程图和代码片段帮助读者理解 Docker 各个功能模块的流程，是学习 Docker 开源系统的良师益友。

——寿黎旦，浙江大学计算机学院教授

近年来，Docker 迅速风靡了云计算世界，但是专门针对 Docker 的技术实现进行深入分析的文章却相对较少。这一方面由于 Docker 技术变化很快，源码分析很快会跟不上版本发展；另一方面，对源代码的解析，需要对整个 Docker 设计具备全局的视角，才能深入浅出地找到源码中的脉络。

宏亮的这本《Docker 源码分析》，恰如其时的出现，弥补了这个空白，对于希望参与到 Docker 社区、参与代码贡献或构建自己的 Docker 应用环境的读者来说，应是一本案头必备书籍。

——王兴宇，《Linux 中国》创始人

在崇尚源码至上的工程师文化里，文档介绍、发布会材料都是苍白的，唯有研读源码，才能深刻理解软件背后的原理。与所有其他软件一样，读源码并不是学习 Docker 最快的途

目发展速度非常快，这次在文章连载内容的基础上出书，为了保证内容的准确性和时效性，宏亮补充了大量 Docker 最新项目的内容，特别是对 Swarm、Machine 和 Compose 这三个模块的开发进展做了紧密的追踪。

这是一本从架构和代码角度讲解 Docker 底层实现的技术图书，我从连载第一篇开始就对这个系列的文章保持了紧密的关注，也目睹了宏亮在后期整理加工成书过程中的辛勤努力。在《 Docker 源码分析》成书付梓出版之际，非常幸运，能够为宏亮写着一篇推荐序。这本书非常适合以下三类读者学习和阅读。

❑ 希望以 Docker 容器交付软件的程序员。

Docker 是未来互联网软件的交付件，这件事随着 OCP 标准的制定，正在逐渐成为事实。程序员和运维工程师都需要了解 Docker 的工作方式，尤其是 Docker 镜像的结构，软件通过 Dockerfile 打包时的优化方式等，这些内容在本书中都有非常详细的阐述。

❑ Docker 化云计算平台的建设者和维护者。

Docker 公司在今年的全球开发者大会上提出了“ Production Ready”的口号，有越来越多的互联网公司和传统企业采用 Docker 来构建开发、测试和运维平台。Docker 在网络、存储、安全等领域的细节，是平台建设者和维护者必须深入了解的部分，这些领域还在不断变化，新的项目也层出不穷，但本书对网络、存储和安全的基本知识和概念，做了非常清晰的介绍，也深入到了底层实现的代码。

❑ Go 语言程序员。

即使不在项目中使用 Docker，本书也能够为 Go 语言程序员带来帮助。Docker 项目中大量采用了 Go 语言，尤其是在处理并发场景时，Docker 对 Go 语言的运用可谓出神入化。本书可以帮助 Go 语言程序员亲身体验特大型项目中 Go 语言的威力，以及实战场景中 Golang 模式和功能的用法。

最后，预祝宏亮在 Docker 的学习和工作中再创佳绩，也希望读者可以从本书收获知识，开阔眼界。

喻　勇

2015 年 7 月 13 日

Preface 前　言

Docker 是什么

Docker 从 2013 年诞生，短短两年时间就在全球 IT 技术圈内迅速走红，实乃技术圈内不可忽视的一阵飓风。然而，Docker 是什么，Docker 带来了什么？

Docker 官方如此描述 Docker：“Build, Ship, Run. An open platform for distributed applications for developers and sysadmins”。换言之，Docker 为开发者与系统管理者提供了分布式应用的开放平台，从而可以便捷地构建、迁移与运行分布式应用。

多年来，IT 行业中开发与运维一直是两个界限清晰的词。开发工程师专门从事软件的开发工作，最终交付软件代码；运维工程师则部署前者交付的软件，并接管软件的运行与管理。然而，在长时间的实践过程中，开发与运维分离的方式难免存在弊病，两者职责的过分清晰导致软件效率的降低。随着分布式系统的流行，系统规模越来越大，软件越来越复杂，系统环境配置暴露的问题层出不穷。究其缘由，还是因为开发人员缺少软件运行环境的认知，而运维人员对软件逻辑所知甚少。在这样的背景下，DevOps 横空出世，提倡开发与运维不可分割，协调并进。

Docker 无疑是 DevOps 大潮中最具实践价值的不二法宝。Docker 从 Linux 内核的角度出发，属于轻量级虚拟化技术，有能力秒级提供应用隔离环境，完成云计算时代分布式应用的第一需求“隔离”。另外，Docker 的镜像技术利用联合文件系统的优势，自下至上打包系统软件、系统环境以及软件程序，将运行环境与应用程序灵活地结合，快速运行 Docker 化的应用程序。同时，可读性极强的 Dockerfile，极大地简化镜像的复杂性，并为镜像的转移与重新构建提供了可能性。

Docker 提供轻便的资源分配方式，解决应用运行与系统环境的依赖，弥合应用跨节点迁移的鸿沟，种种特性都表明 Docker 几乎就是为“云计算”而生的。如今，Docker 社区不断扩大并健康发展，多家国际 IT 巨头也纷纷宣布支持 Docker，这一切更是让全球 IT 人士对 Docker 的未来充满信心。

本书的内容

本书是一本引导读者了解 Docker 实现原理的技术普及书。笔者一直坚信，了解软件或者系统最直接、最透彻的方式就是研读它们的源码。“源码即文档”也是鼓励开发者能更多地从源码的角度去学习软件或系统的本质。

本书的内容主要集中于 3 个部分：Docker 的架构，Docker 的模块，Docker 的三驾马车 Swarm、Machine 以及 Compose。

第一部分，主要从宏观的角度和读者一起领略 Docker 的架构设计，并初步介绍架构中各模块的职责。

第二部分是本书的主体部分，主要针对 Docker 中多个重要的模块进行具体深入分析，包括：Docker Client、Docker Daemon、Docker Server、Docker 网络、Docker 镜像、Docker 容器等。读者可以发现，Docker 的模块之间耦合度非常低，很适合循序渐进，层层深入。第 2 章至第 8 章主要从 Docker 软件的架构入手，勾勒骨架；第 9 章至第 11 章重点讨论 Docker 镜像技术，夯实基础；第 12 章至第 14 章则进一步分析 Docker 容器的始末，阐述本质。

第三部分介绍 Docker 生态三驾马车 Swarm、Machine、Compose。Docker 拥有强大的单机能力，三驾马车可以很好地补充 Docker 的跨主机能力以及部署能力。读者可以通过第 15 章至第 17 章感受 Docker 生态圈中其他功能强大的软件。

希望本书能够让读者最大化地感受 Docker 的神奇与魅力。

关于勘误

由于时间与水平都比较有限，因此本书难免会存在一些纰漏和错误。如果读者发现了问题，请及时与我联系。我也会在本书后续的版本中加以改正。我的邮箱是：allen.sun@daocloud.io。我非常希望和大家一起学习与讨论 Docker，并共同推动 Docker 在社区的发展。

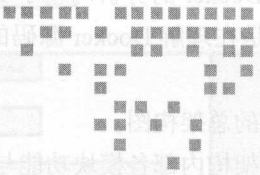
致谢

最后，向本书编写过程中给予我巨大帮助的人们表示最诚挚的感谢。感谢我的父母，没有他们的鼓励和支持，此书不可能在如此短的时间内完成。感谢我的母校浙江大学以及 SEL 实验室的老师与同学们，是他们在我求学过程中给予无私的指引与帮助。感谢我的同事熊中祥，是他在本书编写过程中提出了很多宝贵的建议，尤其在 Machine 和 Compose 部分。感谢我的同事喻勇和冯钊，他们为本书的编写做了很多沟通与协调工作。最后，还要感谢华章公司的编辑们，她们认真细致的工作，使本书以完美的形式展现给各位读者。

孙宏亮

2015 年 6 月

Contents 目 录



第 1 章 Chapter 1

Docker 架构

1.1 引言

Docker 是 Linux 平台上的一款轻量级虚拟化容器的管理引擎。在全球范围内，Docker 还是一个开源项目，整个项目基于 Go 语言开发，代码托管于 GitHub 网站上，并遵从 Apache 2.0 协议。目前，Docker 可以帮助用户在容器内部快速自动化部署应用，并利用 Linux 内核特性命名空间（namespaces）及控制组（cgroups）等为容器提供隔离的运行环境。Docker 借助操作系统层的虚拟化实现资源的隔离，因此 Docker 容器在运行时与虚拟机（VM）的运行有很大的区别，Docker 容器与宿主机共享同一个操作系统，不会有额外的操作系统开销。这样的优势很明显，因而大大提高了资源利用率，并且提升了 I/O 等方面的性能。

众多新颖的特性以及项目本身的开放性，导致 Docker 在不到两年的时间里迅速获得了诸多厂商的青睐，其中更是包括 Google、Microsoft、VMware 等行业领航者。Google 在 2014 年 6 月推出了 Kubernetes，宣布支持 Docker 容器的调度管理，而 2014 年 8 月 Microsoft 则宣布 Azure 上支持 Kubernetes，随后传统虚拟化巨头 VMware 宣布与 Docker 强强合作。2014 年 9 月中旬，Docker 更是获得 4000 万美元的 C 轮融资，以推动分布式应用的发展。

目前，Docker 的前景被普遍看好。未来的云计算领域乃至整个 IT 领域，Docker 都必将扮演不可或缺的角色。为了帮助大家更好地认识 Docker、理解 Docker 并掌握 Docker，本书从 Docker 源码的角度出发，详细介绍 Docker 的架构、Docker 的运行以及 Docker 的特性。本章主要介绍 Docker 架构。

本书关于 Docker 的分析均基于 Docker 1.2.0 版本的源码。

本章目的是在理解 Docker 源码的基础上分析 Docker 架构，分析过程主要按照以下三个部分进行：

- Docker 的总架构图。
- Docker 架构内部各模块功能与实现的分析。
- 通过具体的 Docker 命令，阐述 Docker 的运行流程。

1.2 Docker 总架构图

作为 Linux 平台上的一种容器的管理引擎，Docker 并不像其他大型分布式系统那样复杂。Docker 的源码总量并不多，而且清晰的源码结构使得 Docker 的学习成本并不高。换言之，Docker 源码的学习过程并不枯燥，我们可以从中学到很多东西，如 Go 语言的运用、Docker 架构的设计原理等。Docker 对用户而言是一个简单的 C/S 架构，用户通过客户端与服务器端建立通信，而 Docker 的后端是一个松耦合的架构，架构中的模块各司其职、有机组合，支撑着 Docker 运行。

Docker 的总架构如图 1-1 所示。架构中主要的模块有：DockerClient、DockerDaemon、Docker Registry、Graph、Driver、libcontainer 以及 Docker Container。

对用户而言，Docker Client 是与 Docker Daemon 建立通信的最佳途径。用户通过 Docker Client 发起容器的管理请求，请求最终发往 Docker Daemon。

Docker Daemon 作为 Docker 架构中的主体部分，首先具备服务端的功能，有能力接收 Docker Client 发起的请求；其次具备 Docker Client 请求的处理能力。Docker Daemon 内部所有的任务均由 Engine 来完成，且每一项工作都以一个 Job 的形式存在。

Docker Daemon 需要完成的任务很多，因此 Job 的种类也很多。若用户需要下载容器镜像，Docker Daemon 则会创建一个名为“pull”的 Job，运行时从 Docker Registry 中下载镜像，并通过镜像管理驱动 graphdriver 将下载的镜像存储在 graph 中；若用户需要为 Docker 容器创建网络环境，Docker Daemon 则会创建一个名“allocate_interface”的 Job，通过网络驱动 networkdriver 分配网络接口的资源……

libcontainer 是一套独立的容器管理解决方案，这套解决方案涉及了大量 Linux 内核方面的特性，如：namespaces、cgroups 以及 capabilities 等。libcontainer 很好地抽象了 Linux 的内核特性，并提供完整、明确的接口给 Docker Daemon。

当用户执行运行容器这个命令之后，一个 Docker 容器就处于运行状态，该容器拥有隔离的运行环境、独立的网络栈资源以及受限的资源等。

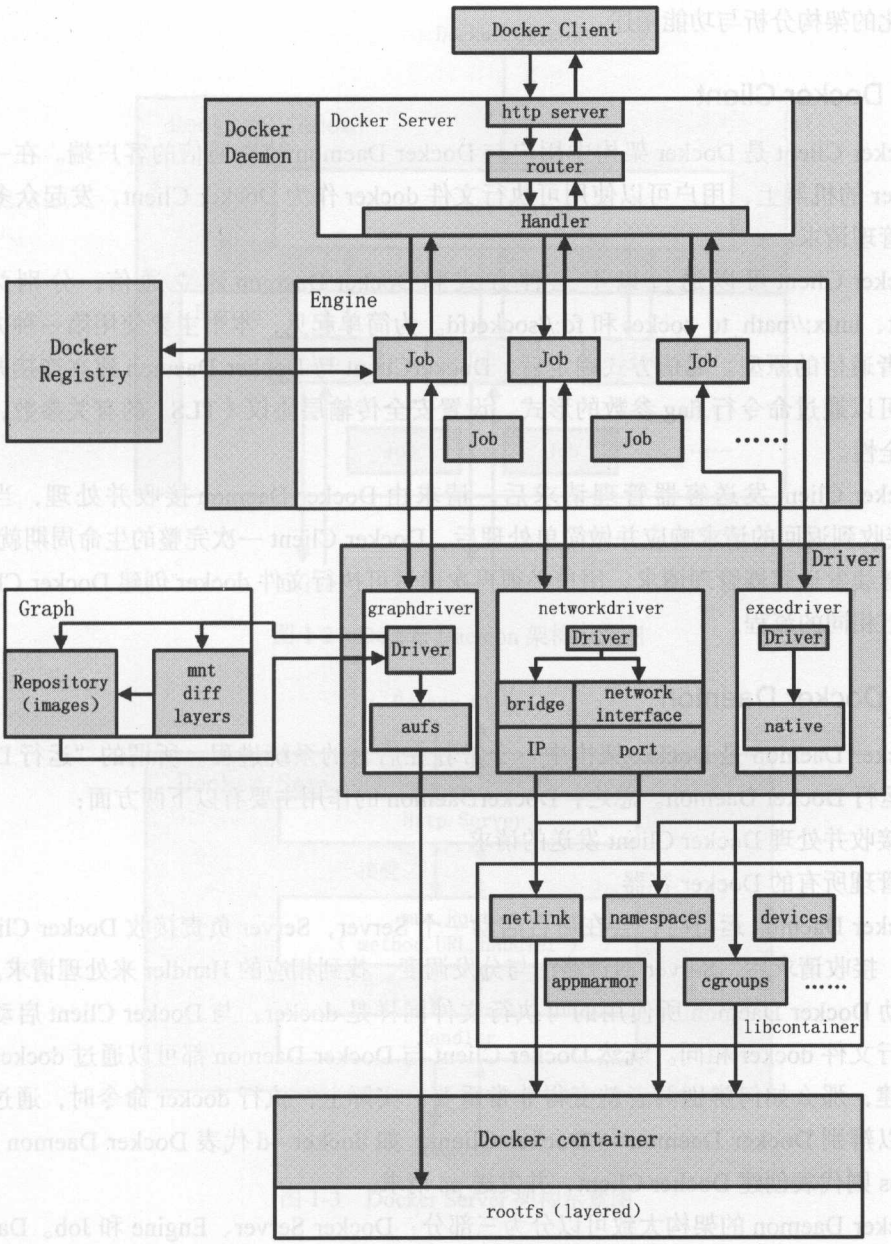


图 1-1　Docker 总架构图

1.3　Docker 各模块功能与实现分析

下面我们将从 Docker 的总架构图入手，抽离出架构内的各个模块，并对各个模块进行

更为细化的架构分析与功能阐述。

1.3.1 Docker Client

Docker Client 是 Docker 架构中用户与 Docker Daemon 建立通信的客户端。在一台安装有 Docker 的机器上，用户可以使用可执行文件 docker 作为 Docker Client，发起众多 Docker 容器的管理请求。

Docker Client 可以通过以下三种方式和 Docker Daemon 建立通信，分别为：tcp://host:port、unix://path_to_socket 和 fd://socketfd。为简单起见，本书主要使用第一种方式作为讲述两者通信的原型。通信方式确定后，DockerClient 与 Docker Daemon 建立连接并传输请求时，可以通过命令行 flag 参数的形式，设置安全传输层协议（TLS）的有关参数，保证传输的安全性。

Docker Client 发送容器管理请求后，请求由 Docker Daemon 接收并处理，当 Docker Client 接收到返回的请求响应并做简单处理后，Docker Client 一次完整的生命周期就此结束。若需要继续发送容器管理请求，用户必须再次通过可执行文件 docker 创建 Docker Client，并走完以上相同的流程。

1.3.2 Docker Daemon

Docker Daemon 是 Docker 架构中一个常驻在后台的系统进程。所谓的“运行 Docker”，即代表运行 Docker Daemon。总之，DockerDaemon 的作用主要有以下两方面：

- 接收并处理 Docker Client 发送的请求。
- 管理所有的 Docker 容器。

Docker Daemon 运行时，会在后台启动一个 Server，Server 负责接收 Docker Client 发送的请求；接收请求后，Server 通过路由与分发调度，找到相应的 Handler 来处理请求。

启动 Docker Daemon 所使用的可执行文件同样是 docker，与 Docker Client 启动所使用的可执行文件 docker 相同。既然 Docker Client 与 Docker Daemon 都可以通过 docker 二进制文件创建，那么如何辨别两者就变得非常重要。实际上，执行 docker 命令时，通过传入的参数可以辨别 Docker Daemon 与 Docker Client，如 docker –d 代表 Docker Daemon 的启动，docker ps 则代表创建 Docker Client，并发送 ps 请求。

Docker Daemon 的架构大致可以分为三部分：Docker Server、Engine 和 Job。Daemon 的架构如图 1-2 所示。

1. Docker Server

Docker Server 在 Docker 架构中专门服务于 Docker Client，它的功能是接收并调度分发 Docker Client 发送的请求。Docker Server 的架构如图 1-3 所示。

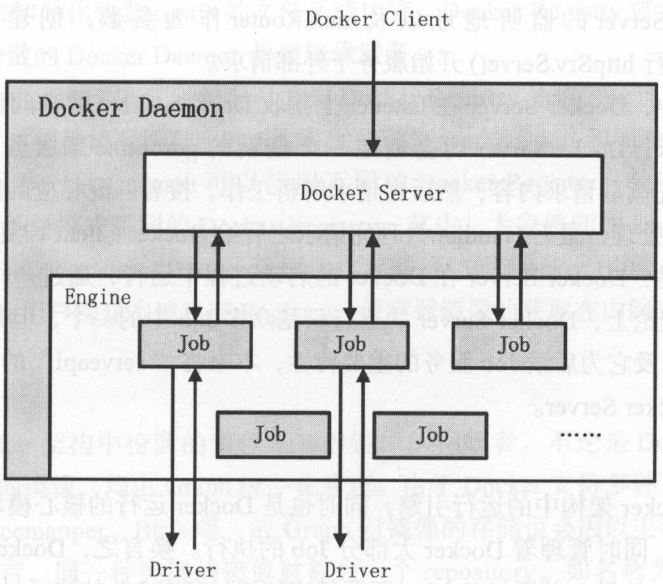


图 1-2　Docker Daemon 架构示意图

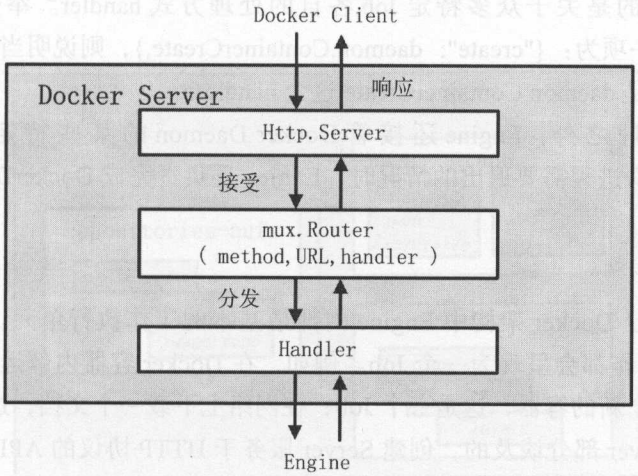


图 1-3　Docker Server 架构示意图

在 Docker Daemon 的启动过程中，DockerServer 第一个完成。Docker Server 通过包 gorilla/mux，创建了一个 mux.Router 路由器，提供请求的路由功能。在 Go 语言中，gorilla/mux 是一个强大的 URL 路由器以及调度分发器。创建路由器之后，Docker Server 为 mux.Router 中添加有效的路由项，每一个路由项由 HTTP 请求方法（PUT、POST、GET 或 DELETE）、URL 和 Handler 三部分组成。

由于 Docker Client 通过 HTTP 协议访问 Docker Daemon，故 DockerServer 创建完 mux.

Router 之后，将 Server 的监听地址以及 mux.Router 作为参数，创建一个 httpSrv=http.Server{}，最终执行 httpSrv.Serve() 开始服务于外部请求。

在服务过程中，Docker Server 在 listener 上接收 Docker Client 的访问请求。对于每一个 Docker Client 请求，DockerServer 均会创建一个全新的 goroutine 来服务。在 goroutine 中，Docker Server 首先读取请求内容，然后做请求解析工作，接着匹配相应的路由项，随后调用相应的 Handler 来处理，最后 Handler 处理完请求之后给 Docker Client 回复响应。

需要注意的是：Docker Server 在 Docker 的启动过程中运行，通过一个名为“serveapi”的 Job 来实现。理论上，Docker Server 的运行只是众多 Job 中的一个，但是为了强调 Docker Server 的重要性以及它为后续 Job 服务的重要特性，本书将“serveapi”的 Job 单独抽离出来分析，理解为 Docker Server。

2. Engine

Engine 是 Docker 架构中的运行引擎，同时也是 Docker 运行的核心模块。Engine 存储着大量的容器信息，同时管理着 Docker 大部分 Job 的执行。换言之，Docker 中大部分任务的执行都需要 Engine 协助，并通过 Engine 匹配相应的 Job 完成 Job 的执行。

在 Docker 源码中，有关 Engine 的数据结构定义中含有一个名为 handlers 的对象。该 handlers 对象存储的是关于众多特定 Job 各自的处理方式 handler。举例说明，Engine 的 handlers 对象中有一项为：{"create": daemon.ContainerCreate,}，则说明当执行名为“create”的 Job 时，执行的是 daemon.ContainerCreate 这个 handler。

除了容器管理之外，Engine 还接管 Docker Daemon 的某些特定任务。当 Docker Daemon 遭遇到自身进程需要退出的情况时，Engine 还负责完成 DockerDaemon 退出前的所有善后工作。

3. Job

Job 可以认为是 Docker 架构中 Engine 内部最基本的工作执行单元。DockerDaemon 可以完成的每一项工作都会呈现为一个 Job。例如，在 Docker 容器内部运行一个进程，这是一个 Job；创建一个新的容器，这是一个 Job；在网络上下载一个文档，这是一个 Job；包括之前在 Docker Server 部分谈及的，创建 Server 服务于 HTTP 协议的 API，这也是一个 Job，等等。

有关 Job 接口的设计，与 UNIX 进程非常相仿。比如说，Job 有一个名称，有运行时参数，有环境变量，有标准输入与标准输出，有标准错误，还有返回状态等。

对于 Job 而言，定义完毕之后，运行才能完成 Job 自身真正的使命。Job 的运行函数 Run() 则用以执行 Job 本身。

1.3.3 Docker Registry

Docker Registry 是一个存储容器镜像（Docker Image）的仓库。容器镜像（Docker Image）

是容器创建时用来初始化容器 rootfs 的文件系统内容。Docker Registry 将大量的容器镜像汇集在一起，并为分散的 Docker Daemon 提供镜像服务。

Docker 的运行过程中，有三种情况可能与 Docker Registry 通信，分别为搜索镜像、下载镜像、上传镜像。这三种情况所对应的 Job 名称分别为 search、pull 和 push。

不同场景下，Docker Daemon 可以使用不同的 Docker Registry。公有 Registry 与私有 Regsitry 就是两种场景模式不同的 Docker Registry。其中，大家熟知的 Docker Hub，就是全球范围内最大的公有 Registry。Docker 可以通过互联网访问 Docker Hub，并下载容器镜像；同时 Docker 也允许用户构建本地私有 Registry，使容器镜像的获取在内网完成。

1.3.4　Graph

Graph 在 Docker 架构中扮演的角色是容器镜像的保管者。不论是 Docker 下载的镜像，还是 Docker 构建的镜像，均由 Graph 统一化管理。由于 Docker 支持多种不同的镜像存储方式，如 aufs、devicemapper、Btrfs 等，故 Graph 对镜像的存储也会因以上种类而存在一些差异。对 Docker 而言，同一种类型的镜像被称为一个 repository，如名称为 ubuntu 的镜像都同属一个 repository；而同一个 repository 下的镜像则会因 tag 存在差异而不同，如 ubuntu 这个 repository 下有 tag 为 12.04 的镜像，也有 tag 为 14.04 的镜像。Docker 中 Graph 的架构如图 1-4 所示。

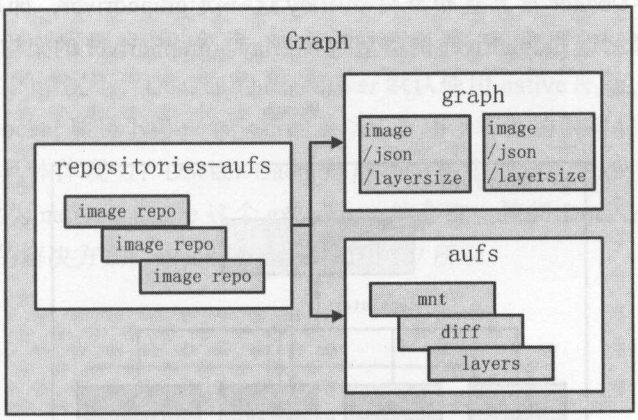


图 1-4　Graph 架构示意图

本书对 Graph 以及容器镜像（Docker Image）的分析将以 aufs 为主，并在第 8 章进行深入分析。

1.3.5　Driver

Driver 是 Docker 架构中的驱动模块。通过 Driver 驱动，Docker 可以实现对 Docker 容器运行环境的定制，定制的维度主要有网络环境、存储方式以及容器执行方式。需要注意的

是，Docker 运行的生命周期中，并非用户所有的操作都是针对 Docker 容器的管理，同时包括用户对 Docker 运行信息的获取，还包括 Docker 对 Graph 的存储与记录等。因此，为了将仅与 Docker 容器有关的管理从 Docker Daemon 的所有逻辑中区分开来，Docker 的创造者设计了 Driver 层来抽象不同类别各自的功能范畴 。

Docker Driver 的实现可以分为以下三类驱动：graphdriver、networkdriver 和 execdriver。

graphdriver 主要用于完成容器镜像的管理，包括从远程 Docker Registry 上下载镜像并进行存储，也包括本地构建完镜像后的存储。当用户下载指定的容器镜像时，graphdriver 将容器镜像分层存储在本地的指定目录下；同时当用户需要使用指定的容器镜像来创建容器时，graphdriver 从本地镜像存储目录中获取指定的容器镜像，并按特定规则为容器准备 rootfs ；另外，当用户需要通过指定 Dockerfile 构建全新镜像时，graphdriver 会负责新镜像的存储管理。

在 graphdriver 的初始化过程之前，有 4 种文件系统或类文件系统的驱动 Driver 在 DockerDaemon 内部注册，它们分别是 aufs、btrfs、vfs 和 devmapper。其中，aufs、btrfs 以及 devmapper 用于容器镜像的管理，vfs 用于容器 volume 的管理。Docker 在初始化之时，优先通过获取系统环境变量“DOCKER_DRIVER”来提取所使用 driver 的指定类型。因此，之后所有的 Graph 操作，都使用该 driver 来执行。Docker 镜像是 Docker 技术中非常关键的。2014 年 12 月，在 Linux 3.18-rc2 版本中 OverlayFS 被合并至 Linux 内核主线，在 Docker 1.4.0 版本发布时，Docker 官方宣布支持 overlay 这一类 graphdriver，即用户在启动 Docker Daemon 时可以选择制定 graphdriver 的类型为 overlay。graphdriver 的架构如图 1-5 所示。

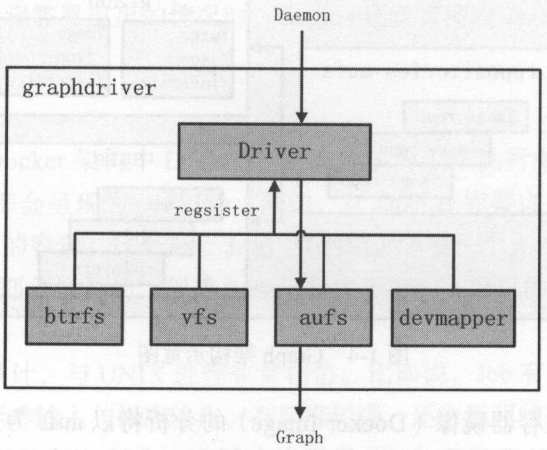


图 1-5　graphdriver 架构示意图

networkdriver 的作用是完成 Docker 容器网络环境的配置，其中包括 Docker Daemon 启动时为 Docker 环境创建网桥；Docker 容器创建前为其分配相应的网络接口资源；以及为 Docker 容器分配 IP、端口并与宿主机做 NAT 端口映射，设置容器防火墙策略等。

networkdriver 的架构如图 1-6 所示。

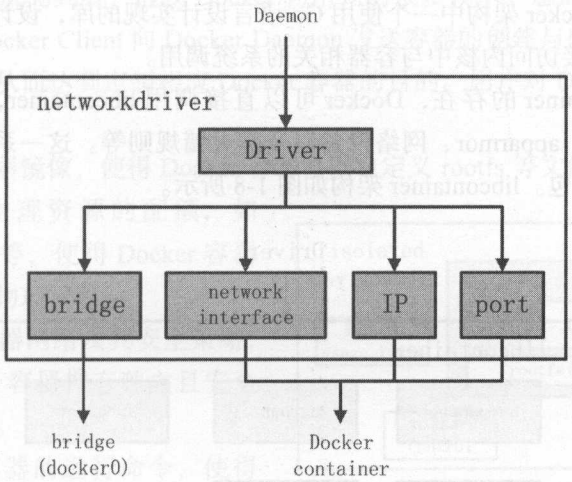


图 1-6　networkdriver 架构示意图

execdriver 作为 Docker 容器的执行驱动，负责创建容器运行时的命名空间，负责容器资源使用的统计与限制，负责容器内部进程的真正运行等。在 Docker 0.9.0 版本之前，execdriver 只能通过 LXC 驱动来实现容器的启动管理。实际上，当时 Docker 通过 LXC 驱动调用 Linux 下的 LXC 工具管理容器的创建，并控制管理容器的生命周期。从 Docker 0.9.0 开始，在继续支持 LXC 的情况下，Docker 的 execdriver 默认使用 native 驱动，native 驱动完全独立于 LXC，属于 Docker 项目下第一个全新的子项目，用于容器的创建与管理。Docker 默认使用 native 驱动的具体体现为：Docker Daemon 启动过程中加载的 ExecDriverflag 参数在配置文件中已经被设为 native。native 这个 execdriver 的存在，使得 Docker 对 Linux 容器的创建与管理有了自己的解决方案。execdriver 架构如图 1-7 所示。

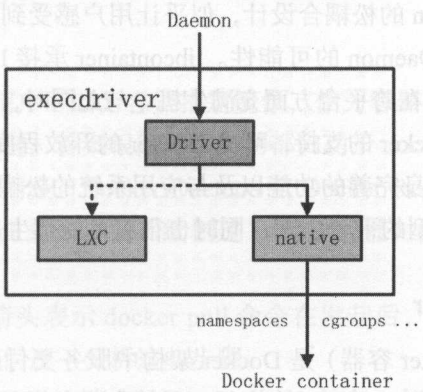


图 1-7　execdriver 架构示意图

1.3.6 libcontainer

libcontainer 是 Docker 架构中一个使用 Go 语言设计实现的库，设计初衷是希望该库可以不依靠任何依赖，直接访问内核中与容器相关的系统调用。

正是由于 libcontainer 的存在，Docker 可以直接调用 libcontainer，而最终操作容器的 namespaces、cgroups、apparmor、网络设备以及防火墙规则等。这一系列操作的完成都不需要依赖 LXC 或者其他包。libcontainer 架构如图 1-8 所示。

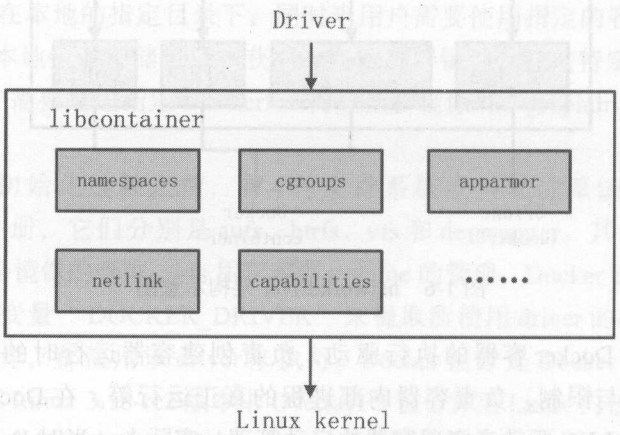


图 1-8 libcontainer 示意图

另外，libcontainer 提供了一整套标准的接口来满足上层对容器管理的需求。或者说，libcontainer 屏蔽了 Docker 上层对容器的直接管理。又由于 libcontainer 使用 Go 这种跨平台的语言开发实现，且本身又可以被上层多种不同的编程语言访问，因此，很难说未来的 Docker 一定会与 Linux 平台紧紧捆绑在一起。Docker Daemon 的逻辑完全有可能位于其他非 Linux 操作系统的平台上，仅仅通过 libcontainer 的远程调用来实现对 Docker 容器的管理。另一方面，libcontainer 与 Docker Daemon 的松耦合设计，似乎让用户感受到了除 Linux Container 之外其他的容器技术接入 Docker Daemon 的可能性。libcontainer 承接 Linux 内核与 Docker Daemon 的同时，也让 Docker 的生态在跨平台方面充满生机。与此同时，Microsoft 在其著名云计算平台 Azure 中，也添加了对 Docker 的支持，可见 Docker 的开放程度与业界的火热度。

暂不谈 Docker，由于本身完善的功能以及与应用系统的松耦合特性，libcontainer 很有可能会在众多其他以容器为原型的平台出现，同时也很有可能催生出云计算领域全新的项目。

1.3.7 Docker Container

Docker Container（Docker 容器）是 Docker 架构中服务交付的最终体现形式。Docker 通过 DockerDaemon 的管理，libcontainer 的执行，最终创建 Docker 容器。Docker 容器作为一个交付单位，功能类似于传统意义上的虚拟机（Virtual Machine），具备资源受限、环境与外

界隔离的特点。然而，实现手段却与 KVM、Xen 等传统虚拟化技术大相径庭。

Docker 容器的从无到有，涉及 Docker 利用到的很多技术。总而言之，用户可以根据自己的需求，通过 Docker Client 向 Docker Daemon 发送容器的创建与启动请求，请求中将携带容器的配置信息，从而达到定制相应 Docker 容器的目的。用户对 Docker 容器的配置有以下 4 个基本方面：

- 通过指定容器镜像，使得 Docker 容器可以自定义 rootfs 等文件系统。
- 通过指定物理资源的配额，如 CPU、内存等，使得 Docker 容器使用受限的物理资源。
- 通过配置容器网络及其安全策略，使得 Docker 容器拥有独立且安全的网络环境。
- 通过指定容器的运行命令，使得 Docker 容器执行指定的任务。

Docker 容器示意图如图 1-9 所示。

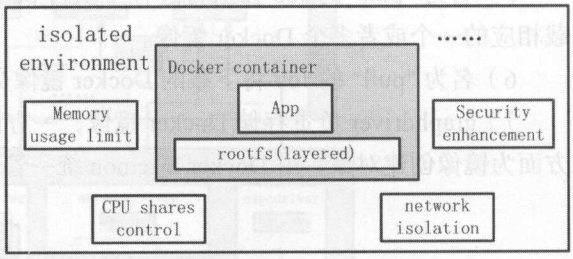


图 1-9　Docker 容器示意图

1.4　Docker 运行案例分析

1.3 节着重介绍了 Docker 架构中各个模块的功能，学完后我们可以对 Docker 的架构有一个宏观的认识。熟悉一款软件，研究一个系统，从静态的角度认识架构的各个模块，仅仅是第一步；从动态的角度，掌握软件或者系统的运行原理，即熟知架构中模块间的通信逻辑，无疑会让自己对软件或系统的理解更上一层楼。本节将从实际的 Docker 运行案例出发，串联 Docker 各模块，从而学习 Docker 的运行流程。分析原型为 Docker 中的 docker pull 与 docker run 两个命令。

1.4.1　docker pull

1.3 节中我们提到，用户可以为容器指定镜像，作为容器运行时的 rootfs，既然如此，镜像从何而来则成为一个关键。答案很简单，一切都归功于 docker pull 命令。

docker pull 命令的作用是：Docker Daemon 从 Docker Registry 下载指定的容器镜像，并将镜像存储在本地的 Graph 中，以备后续创建 Docker 容器时使用。docker pull 命令的执行流程如图 1-10 所示。

图 1-10 中有编号的箭头表示 docker pull 命令在发起后，Docker 架构中相应模块所做的一系列运行操作。下面我们逐一分析这些步骤。

1）Docker Client 处理用户发起的 docker pull 命令，解析完请求以及参数之后，发送一个 HTTP 请求给 Docker Server，HTTP 请求方法为 POST，请求 URL 为 "/images/

create?"+"xxx"，实际意义为下载相应的镜像。

2）Docker Server 接收以上 HTTP 请求，并交给 mux.Router，mux.Router 通过 URL 以及请求方法类型来确定执行该请求的具体 handler。

3）mux.Router 将请求路由分发至相应的 handler，具体为 PostImagesCreate。

4）在 PostImageCreate 这个 handler 之中，创建并初始化一个名为 "pull" 的 Job，之后触发执行该 Job。

5）名为 "pull" 的 Job 在执行过程中执行 pullRepository 操作，即从 Docker Registry 中下载相应的一个或者多个 Docker 镜像。

6）名为 "pull" 的 Job 将下载的 Docker 镜像交给 graphdriver 管理。

7）graphdriver 负责存储 Docker 镜像，一方面将实际镜像存储至本地文件系统中，另一方面为镜像创建对象，由 Docker Daemon 统一管理。

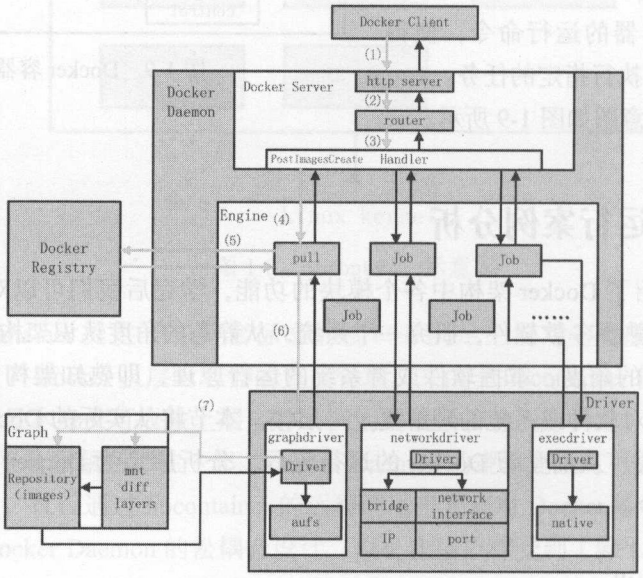


图 1-10　docker pull 命令执行流程示意图

1.4.2　docker run

docker run 命令的作用是创建一个全新的 Docker 容器，并在容器内部运行指定命令。Docker Daemon 处理用户发起的这条命令时，所做工作可以分为两部分：第一，创建 Docker 容器对象，并为容器准备所需的 rootfs；第二，创建容器的运行环境，如网络环境、资源限制等，最终真正运行用户指令。因此，在 docker run 命令的完整执行流程中，Docker Client 给 Docker Server 发送了两次 HTTP 请求，第二次请求的发起取决于第一次请求的返回状态。docker run 命令执行流程如图 1-11 所示。

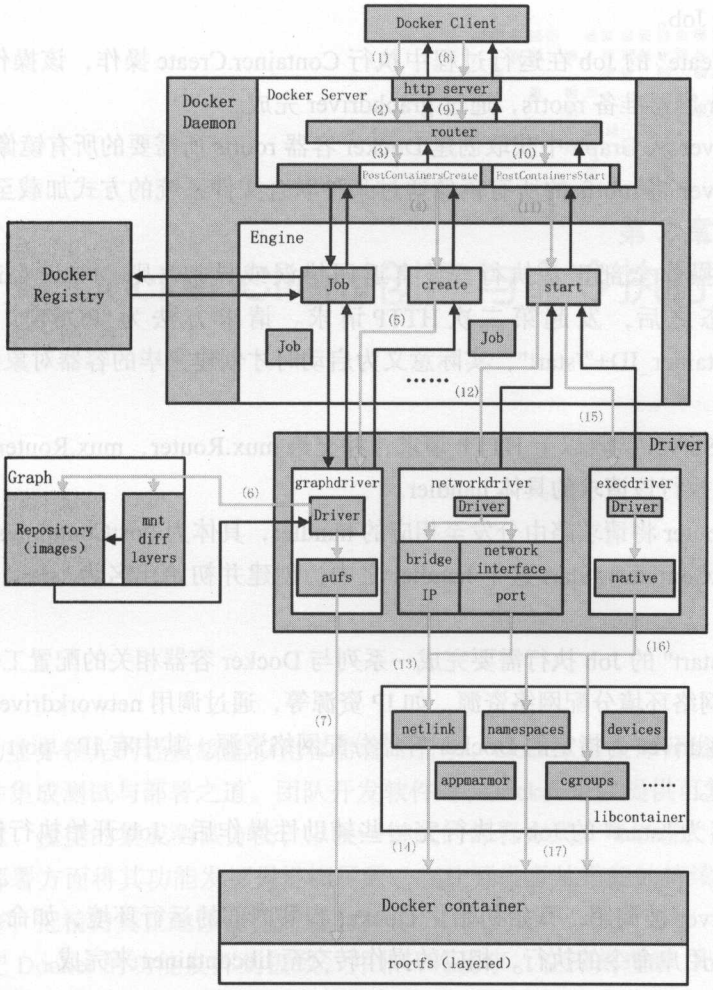


图 1-11 docker run 命令执行流程示意图

图 1-11 中有编号的箭头表示 docker run 命令在发起后，Docker 架构中相应模块所做的一系列运行。下面我们逐一分析这些步骤：

1）Docker Client 处理用户发起的 docker run 命令，解析完请求与参数之后，向 Docker Server 发送一个 HTTP 请求，HTTP 请求方法为 POST，请求 URL 为 "/containers/create?"+"xxx"，实际意义为创建一个容器对象，即 Docker Daemon 程序逻辑中的容器对象，并非实际运行的容器。

2）Docker Server 接收以上 HTTP 请求，并交给 mux.Router，mux.Router 通过 URL 以及请求方法来确定执行该请求的具体 handler。

3）mux.Router 将请求路由分发至相应的 handler，具体为 PostContainersCreate。

4）在 PostContainersCreate 这个 handler 之中，创建并初始化一个名为 "create" 的 Job，

之后触发执行该 Job。

5）名为 "create" 的 Job 在运行过程中执行 Container.Create 操作，该操作需要获取容器镜像来为 Docker 容器准备 rootfs，通过 graphdriver 完成。

6）graphdriver 从 Graph 中获取创建 Docker 容器 rootfs 所需要的所有镜像。

7）graphdriver 将 rootfs 的所有镜像通过某种联合文件系统的方式加载至 Docker 容器指定的文件目录下。

8）若以上操作全部正常执行，没有返回错误或异常，则 Docker Client 收到 Docker Server 返回状态之后，发起第二次 HTTP 请求。请求方法为 "POST"，请求 URL 为 "/containers/"+container_ID+"/start"，实际意义为启动时才创建完毕的容器对象，实现物理容器的真正运行。

9）Docker Server 接收以上 HTTP 请求，并交给 mux.Router，mux.Router 通过 URL 以及请求方法来确定执行该请求的具体 handler。

10）mux.Router 将请求路由分发至相应的 handler，具体为 PostContainersStart。

11）在 PostContainersStart 这个 handler 之中，创建并初始化名为 "start" 的 Job，之后触发执行该 Job。

12）名为 "start" 的 Job 执行需要完成一系列与 Docker 容器相关的配置工作，其中之一是为 Docker 容器网络环境分配网络资源，如 IP 资源等，通过调用 networkdriver 完成。

13）networkdriver 为指定的 Docker 容器分配网络资源，其中有 IP、port 等，另外为容器设置防火墙规则。

14）返回名为 "start" 的 Job，执行完一些辅助性操作后，Job 开始执行用户指令，调用 execdriver。

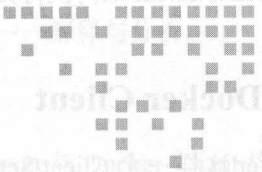
15）execdriver 被调用，开始初始化 Docker 容器内部的运行环境，如命名空间、资源控制与隔离，以及用户命令的执行，相应的操作转交至 libcontainer 来完成。

16）libcontainer 被调用，完成 Docker 容器内部的运行环境初始化，并最终执行用户要求启动的命令。

1.5 总结

本章从 Docker 1.2.0 的源码入手，首先分析抽象出 Docker 的架构图，并对该架构图中的各个模块进行功能与实现的简要分析，最后通过 Docker 的两个基础命令展示了 Docker 内部的运行。

通过对 Docker 架构的学习，可以全面深化对 Docker 设计、功能与价值的理解。同时在借助 Docker 实现用户定制的分布式系统时，也能更好地找到已有平台与 Docker 较为理想的契合点。另外，熟悉 Docker 现有架构以及设计思想，也能为云计算 PaaS 领域带来更多的启示，催生出更多创新想法。



第 2 章 Chapter 2

Docker Client 创建与命令执行

2.1 引言

如今，作为业界领先的轻量级虚拟化容器管理引擎，Docker 给全球开发者提供了一种新颖、便捷的软件集成测试与部署之道。团队开发软件时，Docker 可以提供可复用的运行环境、灵活的资源配置、便捷的集成测试方法，以及一键式的部署方式。可以说，Docker 在简化持续集成、运维部署方面将其功能发挥得淋漓尽致，它让开发者从重复的持续集成、运维部署中完全解放出来，把精力真正地倾注在开发上。

然而，要把 Docker 的功能发挥到极致，并非一件易事。在深刻理解 Docker 架构的情况下，熟练掌握 Docker Client 的使用也非常有必要。前者可以参阅第 1 章，本章主要针对后者，从源码的角度分析 Docker Client，力求帮助开发者更深刻地理解 Docker Client 的具体实现，最终更好地掌握 Docker Client 的使用方法。

本章基于 Docker 1.2.0 的源码，分析 Docker Client 的内容。主要包括两个部分，分别是 DockerClient 的创建与 Docker Client 对命令的执行。两部分分析的具体内容如下。

第一部分分析 Docker Client 的创建。这部分的分析可分为以下三个步骤：

- 分析如何通过 docker 命令，解析出命令行 flag 参数，以及 docker 命令中的请求参数。
- 分析如何处理具体的 flag 参数信息，并收集 Docker Client 所需的配置信息。
- 分析如何创建一个 Docker Client。

第二部分在已有 Docker Client 的基础上，分析如何执行 docker 命令。这部分的分析又可分为以下两个步骤。

- 分析如何解析 docker 命令中的请求参数，获取相应请求的类型。

- 分析 Docker Client 如何执行具体的请求命令，最终将请求发送至 Docker Server。

2.2 创建 Docker Client

对于 Docker 这样一个 Client/Server 的架构，客户端的存在意味着 Docker 相应任务的发起。用户首先需要创建一个 DockerClient，随后将特定的请求类型与参数传递至 Docker Client，最终由 Docker Client 转义成 Docker Server 能识别的形式，并发送至 Docker Server。

Docker Client 的创建实质上是 Docker 用户通过二进制可执行文件 docker，创建与 Docker Server 建立联系的客户端。以下分 3 个小节分别阐述 Docker Client 的创建流程。

Docker Client 完整的运行流程如图 2-1 所示。

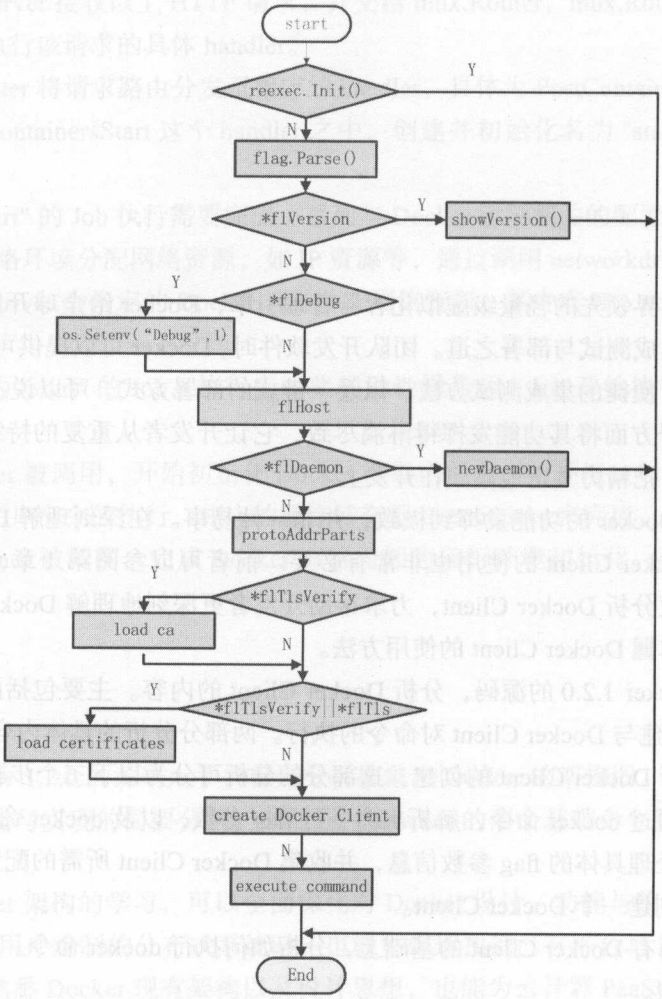


图 2-1 DockerClient 的运行流程

通过学习图 2-1，我们可以更为清晰地了解 Docker Client 创建及执行请求的过程。其中涉及诸多 Docker 源码层次中的专有名词，本章后续会一一解释与分析。

2.2.1　Docker 命令的 flag 参数解析

众所周知，在 Docker 的具体实现中，Docker Server 与 Docker Client 均由可执行文件 docker 来完成创建并启动。那么，了解 docker 可执行文件通过何种方式来区分到底是 Docker Server 还是 Docker Client，就显得尤为重要。

首先通过 docker 命令举例说明其中的区别。Docker Server 的启动，命令为 docker -d 或 docker --daemon=true ；而 Docker Client 的启动则体现为 docker --daemon=false ps、docker pull NAME 等。

其实，对于 Docker 请求中的参数，我们可以将其分为两类：第一类为命令行参数，即 docker 程序运行时所需提供的参数，如：-D、--daemon=true、--daemon=false 等；第二类为 docker 发送给 Docker Server 的实际请求参数，如：ps、pull NAME 等。

对于第一类，我们习惯将其称为 flag 参数，在 Go 语言的标准库中，专门为该类参数提供了一个 flag 包，方便进行命令行参数的解析。

清楚 docker 二进制文件的使用以及基本的命令行 flag 参数之后，我们可以进入实现 Docker Client 创建的源码中，位于 ./docker/docker/docker.go。这个 go 文件包含了整个 Docker 的 main 函数，也就是整个 Docker（不论 Docker Daemon 还是 Docker Client）的运行入口。部分 main 函数代码如下：

```
func main() {
    if reexec.Init() {
        return
    }
    flag.Parse()
    // FIXME: validate daemon flags here
    ...
}
```

以上源码实现中，首先判断 reexec.Init() 方法的返回值，若为真，则直接退出运行，否则将继续执行。reexec.Init() 函数的定义位于 ./docker/reexec/reexec.go，可以发现由于在 docker 运行之前没有任何 Initializer 注册，故该代码段执行的返回值为假。reexec 存在的作用是：协调 execdriver 与容器创建时 dockerinit 这两者的关系。第 13 章在分析 dockerinit 的启动时，将详细讲解 reexec 的作用。

判断 reexec.Init() 之后，Docker 的 main 函数通过调用 flag.Parse() 函数，解析命令行中的 flag 参数。如果熟悉 Go 语言中的 flag 参数，一定知道解析 flag 参数的值之前，程序必须先定义相应的 flag 参数。进一步查看 Docker 的源码，我们可以发现 Docker 在 ./docker/docker/flag.go 中定义了多个 flag 参数，并通过 init 函数进行部分 flag 参数的初始化。代码如下：

```
var (
    flVersion = flag.Bool([]string{"v", "-version"}, false, "Print version
    information and quit")
    flDaemon = flag.Bool([]string{"d", "-daemon"}, false, "Enable daemon mode")
    flDebug = flag.Bool([]string{"D", "-debug"}, false, "Enable debug mode")
    flSocketGroup = flag.String([]string{"G", "-group"}, "docker", "Group to
    assign the unix socket specified by -H when running in daemon modeuse '' (the
    empty string) to disable setting of a group")
    flEnableCors = flag.Bool([]string{"#api-enable-cors", "-api-enable-cors"},
    false, "Enable CORS headers in the remote API")
    flTls = flag.Bool([]string{"-tls"}, false, "Use TLS; implied by tls-verify
    flags")
    flTlsVerify = flag.Bool([]string{"-tlsverify"}, false, "Use TLS and verify
    the remote (daemon: verify client, client: verify daemon)")

    // these are initialized in init() below since their default values depend
    on dockerCertPath which isn't fully initialized until init() runs
    flCa    *string
    flCert  *string
    flKey   *string
    flHosts []string
    )

    func init() {
    flCa = flag.String([]string{"-tlscacert"}, filepath.Join(dockerCertPath,
    defaultCaFile), "Trust only remotes providing a certificate signed by the CA
    given here")
    flCert = flag.String([]string{"-tlscert"}, filepath.Join(dockerCertPath,
    defaultCertFile), "Path to TLS certificate file")
    flKey = flag.String([]string{"-tlskey"}, filepath.Join(dockerCertPath,
    defaultKeyFile), "Path to TLS key file")
    opts.HostListVar(&flHosts, []string{"H", "-host"}, "The socket(s) to bind to
    in daemon mode\nspecified using one or more tcp://host:port, unix:///path/
    to/socket, fd://* or fd://socketfd.")
}
```

以上源码展示了 Docker 如何定义 flag 参数，以及在 init 函数中实现部分 flag 参数的初始化。Docker 的 main 函数执行前，这些变量创建以及初始化工作已经全部完成。这里涉及了 Go 语言的一个特性，即 init 函数的执行。Go 语言中引入其他包（import package）、变量的定义、init 函数以及 main 函数这四者的执行顺序如图 2-2 所示。

关于 Golang 中的 init 函数，深入分析可以得出以下特性：

- ❑ init 函数用于程序执行前包的初始化工作，比如初始化变量等；
- ❑ 每个包可以有多个 init 函数；
- ❑ 包的每一个源文件也可以有多个 init 函数；
- ❑ 同一个包内的 init 函数的执行顺序没有明确的定义；
- ❑ 不同包的 init 函数按照包导入的依赖关系决定初始化的顺序；

❑ init 函数不能被调用，而是在 main 函数调用前自动被调用。

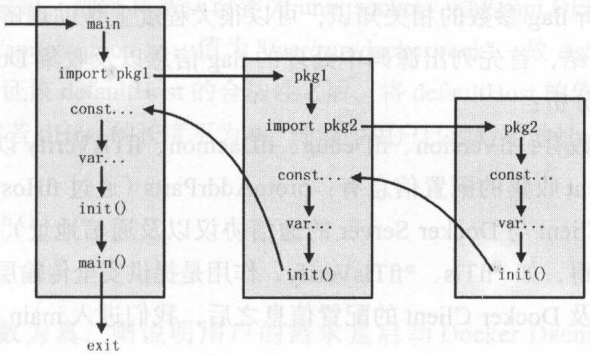


图 2-2　Go 语言程序加载顺序图

清楚 Go 语言一些基本的特性之后，回到 Docker 中来。Docker 的 main 函数执行之前，Docker 已经定义了诸多 flag 参数，并对很多 flag 参数进行初始化。定义并初始化的命令行 flag 参数有：flVersion、flDaemon、flDebug、flSocketGroup、flEnableCors、flTls、flTlsVerify、flCa、flCert、flKey、flHosts 等。

以下具体分析 flDaemon：

❑ 定义：flDaemon = flag.Bool([]string{"d", "-daemon"}, false, "Enable daemon mode");

❑ flDaemon 的类型为 Bool 类型；

❑ flDaemon 名称为 "d" 或者 "-daemon"，该名称会出现在 docker 命令中，如 docker –d;

❑ flDaemon 的默认值为 false；

❑ flDaemon 的用途信息为 "Enable daemon mode"；

❑ 访问 flDaemon 的值时，使用指针 *flDaemon 解引用访问。

在解析命令行 flag 参数时，以下语句为合法的（以 flDaemon 为例）：

❑ -d, --daemon

❑ -d=true, --daemon=true

❑ -d="true", --daemon="true"

❑ -d='true', --daemon='true'

当解析到第一个非定义的 flag 参数时，命令行 flag 参数解析工作结束。举例说明，当执行 docker 命令 docker --daemon=false --version=false ps 时，flag 参数解析主要完成两个工作：

❑ 完成命令行 flag 参数的解析，根据 flag 的名称 -daemon 和 -version，得知具体的 flag 参数为 flDaemon 和 flVersion，并获得相应的值，均为 false。

❑ 遇到第一个非定义的 flag 参数 ps 时，flag 包会将 ps 及其之后所有的参数存入 flag.Args()，以便之后执行 Docker Client 具体的请求时使用。

如需深入学习 flag 的实现，可以参见 Docker 源码 ./docker/pkg/mflag/flag.go。

2.2.2 处理 flag 信息并收集 Docker Client 的配置信息

理解 Go 语言解析 flag 参数的相关知识，可以很大程度上帮助理解 Docker 的 main 函数的执行流程。通过总结，首先列出源码中处理的 flag 信息以及收集 Docker Client 的配置信息，然后再一一进行分析：

- 处理的 flag 参数有：flVersion、flDebug、flDaemon、flTlsVerify 以及 flTls。
- 为 Docker Client 收集的配置信息有：protoAddrParts（通过 flHosts 参数获得，作用是提供 Docker Client 与 Docker Server 的通信协议以及通信地址）、tlsConfig（通过一系列 flag 参数获得，如 *flTls、*flTlsVerify，作用是提供安全传输层协议的保障）。

清楚 flag 参数以及 Docker Client 的配置信息之后，我们进入 main 函数的源码，具体分析如下。

在 flag.Parse() 之后的源码如下：

```
if *flVersion {
    showVersion()
    return
}
```

以上代码很好理解，解析 flag 参数后，若 Docker 发现 flag 参数 flVersion 为真，则说明 Docker 用户希望查看 Docker 的版本信息。此时，Docker 调用 showVersion() 显示版本信息，并从 main 函数退出；否则的话，继续往下执行。

```
if *flDebug {
    os.Setenv("DEBUG", "1")
}
```

若 flDebug 参数为真的话，Docker 通过 os 包中的 Setenv 函数创建一个名为 DEBUG 的环境变量，并将其值设为 "1"；继续往下执行。

```
if len(flHosts) == 0 {
    defaultHost := os.Getenv("DOCKER_HOST")
    if defaultHost == "" || *flDaemon {
        // If we do not have a host, default to unix socket
        defaultHost = fmt.Sprintf("unix://%s", api.DEFAULTUNIXSOCKET)
    }
    if _, err := api.ValidateHost(defaultHost); err != nil {
        log.Fatal(err)
    }
    flHosts = append(flHosts, defaultHost)
}
```

以上的源码主要分析内部变量 flHosts。flHosts 的作用是为 Docker Client 提供所要连接的 host 对象，也就是为 Docker Server 提供所要监听的对象。

在分析过程中，首先判断 flHosts 变量是否长度为 0。若是的话，则说明用户并没有显性传入地址，此时 Docker 的策略为选用默认值。Docker 通过 os 包获取名为 DOCKER_HOST 环

境变量的值，将其赋值于 defaultHost。若 defaultHost 为空或者 flDaemon 为真，说明目前还没有一个定义的 host 对象，则将其默认设置为 unix socket，值为 api.DEFAULTUNIXSOCKET，该常量位于 ./docker/api/common.go，值为 "/var/run/docker.sock"，故 defaultHost 为 "unix:///var/run/docker.sock"。验证该 defaultHost 的合法性之后，将 defaultHost 的值追加至 flHost 的末尾，继续往下执行。当然若 flHost 的长度不为 0，则说明用户已经指定地址，同样继续往下执行。

```
if *flDaemon {
    mainDaemon()
    return
}
```

若 flDaemon 参数为真，则说明用户的需求是启动 Docker Daemon。Docker 随即执行 mainDaemon 函数，实现 Docker Daemon 的启动；若 mainDaemon 函数执行完毕，则退出 main 函数。一般 mainDaemon 函数不会主动终结，Docker Daemon 将作为一个常驻进程运行在宿主机上。本章着重介绍 Docker Client 的启动，故假设 flDaemon 参数为假，不执行以上代码块。继续往下执行。

```
if len(flHosts) > 1 {
     log.Fatal("Please specify only one -H")
}
protoAddrParts := strings.SplitN(flHosts[0], "://", 2)
```

由于不执行 Docker Daemon 的启动流程，故属于 Docker Client 的执行逻辑。首先，判断 flHosts 的长度是否大于 1。若 flHosts 的长度大于 1，则说明需要新创建的 Docker Client 访问不止 1 个 Docker Daemon 地址，显然逻辑上行不通，故抛出错误日志，提醒用户只能指定一个 Docker Daemon 地址。接着，Docker 将 flHosts 这个 string 数组中的第一个元素进行分割，通过 "://" 来分割，分割出的两个部分放入变量 protoAddrParts 数组中。protoAddrParts 的作用是：解析出 Docker Client 与 Docker Server 建立通信的协议与地址，为 Docker Client 创建过程中不可或缺的配置信息之一。一般情况下，flHosts[0] 的值可以是 tcp://0.0.0.0:2375 或者 unix:///var/run/docker.sock 等。

```
var (
    cli       *client.DockerCli
    tlsConfig tls.Config
)
tlsConfig.InsecureSkipVerify = true
```

由于之前已经假设过 flDaemon 为假，可以认定 main 函数的运行是为了 Docker Client 的创建与执行。Docker 在这里创建了两个变量：一个为类型是 *client.DockerCli 的对象 cli，另一个为类型是 tls.Config 的对象 tlsConfig。定义完变量之后，Docker 将 tlsConfig 的 InsecureSkipVerify 属性置为真。tlsConfig 对象的创建是为了保障 cli 在传输数据的时候遵循安全传输层协议（TLS）。安全传输层协议（TLS）用于确保两个通信应用程序之间的保密性

与数据完整性。tlsConfig 是 Docker Client 创建过程中可选的配置信息。

```
// If we should verify the server, we need to load a trusted ca
if *flTlsVerify {
    *flTls = true
    certPool := x509.NewCertPool()
    file, err := ioutil.ReadFile(*flCa)
    if err != nil {
        log.Fatalf("Couldn't read ca cert %s: %s", *flCa, err)
    }
    certPool.AppendCertsFromPEM(file)
    tlsConfig.RootCAs = certPool
    tlsConfig.InsecureSkipVerify = false
}
```

若 flTlsVerify 这个 flag 参数为真，则说明 Docker Client 需 Docker Server 一起验证连接的安全性。此时，tlsConfig 对象需要加载一个受信的 ca 文件。该 ca 文件的路径为 *flCA 参数的值，最终完成 tlsConfig 对象中 RootCAs 属性的赋值，并将 InsecureSkipVerify 属性置为假。

```
// If tls is enabled, try to load and send client certificates
if *flTls || *flTlsVerify {
    _, errCert := os.Stat(*flCert)
    _, errKey := os.Stat(*flKey)
    if errCert == nil && errKey == nil {
        *flTls = true
        cert, err := tls.LoadX509KeyPair(*flCert, *flKey)
        if err != nil {
            log.Fatalf("Couldn't load X509 key pair: %s. Key encrypted?", err)
        }
        tlsConfig.Certificates = []tls.Certificate{cert}
    }
}
```

如果 flTls 和 flTlsVerify 两个 flag 参数中有一个为真，则说明需要加载并发送客户端的证书。最终将证书内容交给 tlsConfig 的 Certificates 属性。

至此，flag 参数已经全部处理完毕，DockerClient 也已经收集到所需的配置信息。下一节将主要分析如何创建 Docker Client。

2.2.3 如何创建 Docker Client

Docker Client 的创建其实就是在已有配置参数信息的情况下，通过 Client 包中的 NewDockerCli 方法创建一个 Docker Clinet 实例 cli。具体源码实现如下：

```
if *flTls || *flTlsVerify {
    cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr,
    protoAddrParts[0], protoAddrParts[1], &tlsConfig)
} else {
    cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr,
```

```
        protoAddrParts[0], protoAddrParts[1], nil)
    }
```

若 flag 参数 flTls 为真或者 flTlsVerify 为真，则说明需要使用 TLS 协议来保障传输的安全性，故创建 Docker Client 的时候，将 tlsConfig 参数传入；否则，同样创建 Docker Client，只不过 tlsConfig 为 nil。

关于 Client 包中的 NewDockerCli 函数的实现，可以具体参见 ./docker/api/client/cli.go。

```
func NewDockerCli(in io.ReadCloser, out, err io.Writer, proto, addr string,
tlsConfig *tls.Config) *DockerCli {
var (
    isTerminal = false
    terminalFd uintptr
    scheme     = "http"
)

if tlsConfig != nil {
    scheme = "https"
}

if in != nil {
    if file, ok := out.(*os.File); ok {
        terminalFd = file.Fd()
        isTerminal = term.IsTerminal(terminalFd)
    }
}

if err == nil {
    err = out
}
return &DockerCli{
    proto:        proto,
    addr:         addr,
    in:           in,
    out:          out,
    err:          err,
    isTerminal: isTerminal,
    terminalFd: terminalFd,
    tlsConfig:   tlsConfig,
    scheme:       scheme,
    }
}
```

总体而言，创建 DockerCli 对象的过程比较简单。较为重要的 DockerCli 的属性有：proto，DockerClient 与 Docker Server 的传输协议；addr，Docker Client 需要访问的 host 目标地址；tlsConfig，安全传输层协议的配置。若 tlsConfig 不为空，则说明需要使用安全传输层协议，DockerCli 对象的 scheme 设置为“ https”，另外还有关于输入、输出以及错误显示的配置等。最终函数返回 DockerCli 对象。

通过调用 client 包中的 NewDockerCli 函数，程序最终创建了 Docker Client，返回 main 包中的 main 函数之后，程序继续往下执行。

2.3 Docker 命令执行

main 函数执行到这个阶段，有以下内容需要为 Docker 命令的执行服务：创建完毕的 Docker Client，docker 命令中的请求参数（经 flag 解析后存放于 flag.Arg()）。也就是说，程序需要使用 Docker Client 来分析 Docker 命令中的请求参数，得出请求的类型，转义为 Docker Server 可以识别的请求之后，最终发送给 Docker Server。

Docker Client 主要完成两方面的工作：解析请求命令，得出请求类型；执行具体类型的请求。本节将从这两个方面深入分析 Docker Client。

2.3.1 Docker Client 解析请求命令

Docker Client 解析请求命令的工作，在 Docker 命令执行部分第一个完成。创建 Docker Client 之后，回到 main 函数中，继续执行的源码如下（位于 ./docker/docker/docker.go#L102-L110）：

```
if err := cli.Cmd(flag.Args()...); err != nil {
    if sterr, ok := err.(*utils.StatusError); ok {
        if sterr.Status != "" {
            log.Println(sterr.Status)
        }
        os.Exit(sterr.StatusCode)
    }
    log.Fatal(err)
}
```

学习以上源码可以发现，正如之前所说，Docker Client 首先解析存放于 flag.Args() 中的具体请求参数，执行的函数为 cli 对象的 Cmd 函数。Cmd 函数的定义如下（位于 ./docker/api/client/cli.go#L51-L61）：

```
// Cmd executes the specified command
func (cli *DockerCli) Cmd(args ...string) error {
    if len(args) > 0 {
        method, exists := cli.getMethod(args[0])
        if !exists {
            fmt.Println("Error: Command not found:", args[0])
            return cli.CmdHelp(args[1:]...)
        }
        return method(args[1:]...)
    }
    return cli.CmdHelp(args...)
}
```

由代码注释可知，Cmd 函数执行具体的指令。在源码实现中，首先判断请求参数列表的长度是否大于 0。若长度不大于 0，则说明没有请求信息，返回 docker 命令的 Help 信息；若长度大于 1，则说明有请求信息，那么 Docker Client 首先通过请求参数列表中的第一个元素 args[0] 来获取具体的请求方法 method。若上述 method 方法不存在，则返回 docker 命令的 Help 信息，若存在，调用具体的 method 方法，参数为 args[1] 及其之后所有的请求参数。

还是以一个具体的 docker 命令为例，docker --daemon=false --version=false pull Image_Name。通过以上 Docker Client 的分析，可以总结出以下执行流程。

1）解析 flag 参数之后，Docker 将 docker 请求参数 "pull" 和 "Image_Name" 存放于 flag.Args()。

2）创建好的 Docker Client 为 cli，cli 执行 cli.Cmd(flag.Args()…)。

3）Cmd 函数中，通过 args[0] 也就是 "pull"，执行 cli.getMethod(args[0])，获取 method 的名称。

4）在 getMothod 方法中，Docker 通过处理最终返回 method 值为 "CmdPull"。

5）最终执行 method(args[1:]…) 也就是 CmdPull(args[1:]…)。

2.3.2　Docker Client 执行请求命令

上一小节通过一系列的命令解析，最终找到了具体的命令执行方法，本小节主要介绍 Docker Client 如何通过具体的执行方法，处理并发送请求。

不同的 Docker 尽管请求内容不同，但是请求执行流程大致相同，故本节依旧以一个例子来阐述其中的流程，例子为：docker pull Image_Name。该命令的作用为：DockerClient 发起下载镜像的请求，最终由 Docker Server 接收请求，由 Docker Daemon 完成镜像的下载与存储。

Docker Client 在执行 docker pull Image_Name 请求命令时，执行 CmdPull 函数，传入参数为 args[1:]...，即 Image_Name。源码实现位于 ./docker/api/client/command.go#L1183-L1224。

下面逐一分析 CmdPull 的源码实现。

```
cmd := cli.Subcmd("pull", "NAME[:TAG]", "Pull an image or a repository from the
registry")
```

通过 cli 包中的 Subcmd 方法定义一个类型为 Flagset 的对象 cmd。

```
tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a
repository")
```

为 cmd 对象定义一个类型为 String 的 flag，名为 "#t" 或 "#-tag"，初始值为空，目前这个 flag 参数基本已经被弃用。

```
if err := cmd.Parse(args); err != nil {
        return nil
}
```

将 args 参数进行第二次 flag 参数解析，解析过程中，先提取出是否有符合 tag 这个 flag 的参数。若有，将其赋值给 tag 参数，其余的参数存入 cmd.NArg()；若没有，则将所有的参数存入 cmd.NArg() 中。

```
if cmd.NArg() != 1 {
    cmd.Usage()
    return nil
}
```

判断经过 flag 解析后的参数列表，若参数列表中参数的个数不为 1，则说明需要下拉多个镜像或者没有指定下拉镜像名称，pull 命令均不支持，则调用错误处理方法 cmd.Usage()，并返回 nil。

```
var (
    v      = url.Values{}
    remote = cmd.Arg(0)
)
v.Set("fromImage", remote)
if *tag == "" {
    v.Set("tag", *tag)
}
```

创建一个 map 类型的变量 v，该变量用于存放下拉镜像时所需的 URL 参数；随后将参数列表的第一个值 cmd.Arg(0) 赋给 remote 变量，并将 remote 作为键将 fromImage 的值添加至 v。

```
remote, _ = parsers.ParseRepositoryTag(remote)
// Resolve the Repository name from fqn to hostname + name
hostname, _, err := registry.ResolveRepositoryName(remote)
if err != nil {
    return err
}
```

通过 remote 变量首先得到镜像的 repository 名称，并赋给 remote 自身，随后通过解析改变后的 remote，得出镜像所在的 host 地址，即 Docker Registry 的地址。若用户没有指定 Docker Registry 的地址，则 Docker 默认地址为 Docker Hub 地址 https://index.docker.io/v1/。

```
cli.LoadConfigFile()
// Resolve the Auth config relevant for this server
authConfig := cli.configFile.ResolveAuthConfig(hostname)
```

通过 cli 对象获取与 Docker Server 通信所需要的认证配置信息。

```
pull := func(authConfig registry.AuthConfig) error {
    buf, err := json.Marshal(authConfig)
    if err != nil {
        return err
    }
    registryAuthHeader := []string{
```

```
            base64.URLEncoding.EncodeToString(buf),
        }
        return cli.stream("POST", "/images/create?"+v.Encode(), nil, cli.out,
        map[string][]string{
        "X-Registry-Auth": registryAuthHeader,
        })
}
```

定义一个名为 pull 的函数，传入的参数类型为 registry.AuthConfig，返回类型为 error。函数执行块中最主要的内容为：cli.stream(……) 部分。该部分具体向 Docker Server 发送 POST 请求，请求的 url 为 "/images/create?"+v.Encode()，请求的认证信息为：map[string][]string{"X-Registry-Auth": registryAuthHeader,}。

```
if err := pull(authConfig); err != nil {
    if strings.Contains(err.Error(), "Status 401") {
        fmt.Fprintln(cli.out, "\nPlease login prior to pull:")
        if err := cli.CmdLogin(hostname); err != nil {
            return err
        }
        authConfig := cli.configFile.ResolveAuthConfig(hostname)
        return pull(authConfig)
    }
    return err
}
```

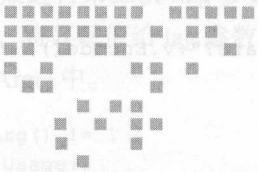
由于上一个步骤只是定义 pull 函数，这一步骤具体调用执行 pull 函数，实现实际意义上的下载请求发送。若返回成功则表明请求完成，程序直接退出，若返回错误，则做相应的错误处理。若返回错误为 401，则表示用户下载的镜像必须用户先登录，随即 Docker Client 转至登录环节，完成之后，继续执行 pull 函数，若完成则最终返回。

以上便是 pull 请求的全部执行过程，其他请求的执行在流程上也是大同小异。总之，请求执行过程中，大多都是将命令行中关于请求的参数进行初步处理，并添加相应的辅助信息，最终通过指定的协议向 Docker Server 发送 Docker Client 和 Docker Server 约定好的 API 请求。

2.4　总结

本章从源码的角度分析了如何通过 docker 可执行文件创建 Docker Client，最终发送用户请求至 Docker Server。

通过学习与理解 Docker Client 相关的源码实现，不仅可以让用户熟练掌握 Docker 命令的使用，还可以使用户在特殊情况下有能力修改 Docker Client 的源码，使其满足自身系统的某些特殊需求，以达到定制 Docker Client 的目的，最大限度地发挥 Docker 开源思想的价值。



Chapter 3

第 3 章 启动 Docker Daemon

3.1 引言

自 Docker 诞生以来，便引领了轻量级虚拟化容器领域的技术热潮。在这一潮流下，Google、IBM、Redhat 等业界翘楚纷纷加入 Docker 阵营。虽然目前 Docker 仍主要基于 Linux 平台，但是 Microsoft 却多次宣布对 Docker 的支持，从先前宣布的 Azure 支持 Docker 与 Kubernetes，到如今宣布的下一代 Windows Server 原生态支持 Docker。Microsoft 的这一系列举措多少喻示着向 Linux 世界的妥协，当然这也不得不让世人对 Docker 的巨大影响力有重新的认识。

Docker 的影响力不言而喻，但如果需要深入学习 Docker 的内部实现，最重要的就是理解 Docker Daemon。在 Docker 架构中，Docker Client 通过特定的协议与 Docker Daemon 进行通信，而 Docker Daemon 主要承载了 Docker 运行过程中的大部分工作。

Docker Daemon 是 Docker 架构中运行在后台的守护进程，大致可以分为 Docker Server、Engine 和 Job 三部分。三者的关系大致如下：Docker Daemon 通过 Docker Server 模块接收 Docker Client 的请求，并在 Engine 中处理请求，然后根据请求类型，创建出指定的 Job 并运行。由于用户的请求不同，DockerDaemon 会创建不同的 Job 来完成任务，如：用户发起镜像下载请求，DockerDaemon 创建名为“pull”的 Job；用户发起启动容器的请求，DockerDaemon 创建名为“start”的 Job……

Docker Daemon 的架构如图 3-1 所示。

本章从源码的角度，主要分析 Docker Daemon 的启动流程。由于 Docker Daemon 和 Docker Client 的启动流程有很多的相似之处，故本章不再赘述 Docker Daemon 启动的前期

工作、flag 参数的解析等内容，着重分析 Docker Daemon 启动流程中最为重要的环节：创建 Daemon 过程中 mainDaemon() 的实现。

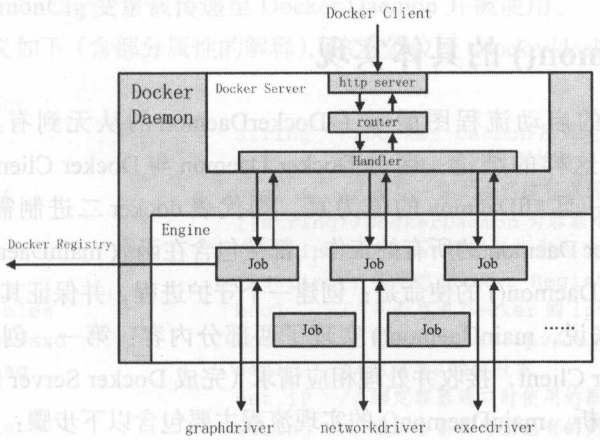


图 3-1　DockerDaemon 架构示意图

3.2　Docker Daemon 的启动流程

Docker Daemon 和 Docker Client 的启动均通过可执行文件 docker 完成，因此两者的启动流程非常相似。Docker 可执行文件运行时，程序运行通过不同的命令行 flag 参数，区分两者，并最终运行两者各自相应的部分。

启动 Docker Daemon 时，一般可以使用以下命令：docker --daemon=true、docker –d; docker -d=true 等。随后由 Docker 的 main() 函数来解析以上命令的相应 flag 参数，并最终完成 Docker Daemon 的启动。

首先，附上 Docker Daemon 的启动流程图，如图 3-2 所示。

本书第 2 章已经描述了 Docker 中 main() 函数运行的很多前期工作，Docker Daemon 的启动也会涉及这些工作，故

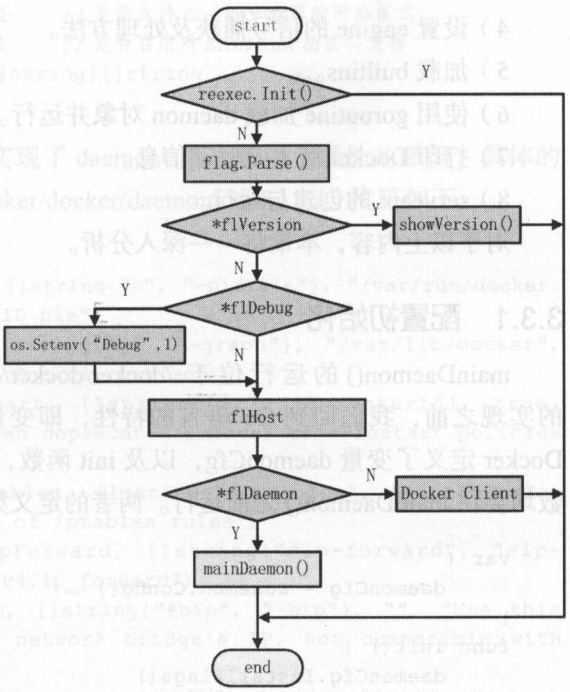


图 3-2　DockerDaemon 启动流程图

在此略去相同部分，主要针对后续仅和 Docker Daemon 相关的内容进行深入分析，即 mainDaemon() 的具体源码实现。

3.3 mainDaemon() 的具体实现

Docker Daemon 的启动流程图展示了 DockerDaemon 的从无到有。通过分析流程图，我们可以得出一个这样的结论：区分 Docker Daemon 与 Docker Client 的关键在于 flag 参数 flDaemon 的值。一旦 *flDaemon 的值为真，则代表 docker 二进制需要启动的是 Docker Daemon。有关 Docker Daemon 的所有的工作，都被包含在函数 mainDaemon() 的具体实现中。

宏观来讲，mainDaemon() 的使命是：创建一个守护进程，并保证其正常运行。

从功能的角度来说，mainDaemon() 实现了两部分内容：第一，创建 Docker 运行环境；第二，服务于 Docker Client，接收并处理相应请求（完成 Docker Server 的初始化）。

从实现细节来分析，mainDaemon() 的实现流程主要包含以下步骤：

1）daemon 的配置初始化。这部分在 init() 函数中实现，即在 mainDaemon() 运行前就执行，但由于这部分内容和 mainDaemon() 的运行息息相关，可以认为是 mainDaemon() 运行的先决条件。

2）命令行 flag 参数检查。

3）创建 engine 对象。

4）设置 engine 的信号捕获及处理方法。

5）加载 builtins。

6）使用 goroutine 加载 daemon 对象并运行。

7）打印 Docker 版本及驱动信息。

8）serveapi 的创建与运行。

对于以上内容，本章将一一深入分析。

3.3.1 配置初始化

mainDaemon() 的运行位于 ./docker/docker/docker/daemon.go，深入分析 mainDaemon() 的实现之前，我们回到 Go 语言的特性，即变量与 init 函数的执行顺序。在 daemon.go 中，Docker 定义了变量 daemonCfg，以及 init 函数，通过 Go 语言的特性，变量的定义与 init 函数均会在 mainDaemon() 之前运行。两者的定义如下：

```
var (
    daemonCfg = &daemon.Config{}
)
func init() {
    daemonCfg.InstallFlags()
}
```

首先，Docker声明一个名为daemonCfg的变量，代表整个Docker Daemon的配置信息。定义完毕之后，init函数的存在，使得daemonCfg变量能获取相应的属性值。在Docker Daemon启动时，daemonCfg变量被传递至Docker Daemon并被使用。

Config对象的定义如下（含部分属性的解释），该对象位于./docker/docker/daemon/config.go：

```
type Config struct {
    Pidfile                    string  //Docker Daemon所属进程的PID文件
    Root                       string  //Docker运行时所使用的root路径
    AutoRestart                bool    //是否一直支持创建容器的重启
    Dns                        []string//DockerDaemon为容器准备的DNS Server地址
    DnsSearch                  []string//Docker使用的指定的DNS查找地址
    Mirrors                    []string//指定的Docker Registry镜像地址
    EnableIptables             bool    //是否启用Docker的iptables功能
    EnableIpForward            bool    //是否启用net.ipv4.ip_forward功能
    EnableIpMasq               bool    //启用IP伪装技术
    DefaultIp                  net.IP  //绑定容器端口时使用的默认IP
    BridgeIface                string  //添加容器网络至已有的网桥接口名
    BridgeIP                   string  //创建网桥的IP地址
    FixedCIDR                  string  //指定IP的IPv4子网，必须被网桥子网包含
    InterContainerCommunication bool   //是否允许宿主机上Docker容器间的通信
    GraphDriver                string  //Docker Daemon运行时使用的特定存储驱动
    GraphOptions               []string//可设置的存储驱动选项
    ExecDriver                 string  //Docker运行时使用的特定exec驱动
    Mtu                        int     //设置容器网络接口的MTU
    DisableNetwork             bool    //是否支持Docker容器的网络模式
    EnableSelinuxSupport       bool    //是否启用对SELinux功能的支持
    Context                    map[string][]string
}
```

Docker声明daemonCfg之后，init函数实现了daemonCfg变量中各属性的赋值，具体的实现为：daemonCfg.InstallFlags()，位于./docker/docker/daemon/config.go，代码如下：

```
func (config *Config) InstallFlags() {
    flag.StringVar(&config.Pidfile, []string{"p", "-pidfile"}, "/var/run/docker.
    pid", "Path to use for daemon PID file")
    flag.StringVar(&config.Root, []string{"g", "-graph"}, "/var/lib/docker",
    "Path to use as the root of the Docker runtime")
    flag.BoolVar(&config.AutoRestart, []string{"#r", "#-restart"}, true,
    "--restart on the daemon has been deprecated infavor of --restart policies
    on docker run")
    flag.BoolVar(&config.EnableIptables, []string{"#iptables", "-iptables"},
    true, "Enable Docker's addition of iptables rules")
    flag.BoolVar(&config.EnableIpForward, []string{"#ip-forward", "-ip-
    forward"}, true, "Enable net.ipv4.ip_forward")
    flag.StringVar(&config.BridgeIP, []string{"#bip", "-bip"}, "", "Use this
    CIDR notation address for the network bridge's IP, not compatible with
    -b")
    flag.StringVar(&config.BridgeIface, []string{"b", "-bridge"}, "", "Attach
    containers to a pre-existing network bridge\nuse 'none' to disable
```

```
        container networking")
        flag.BoolVar(&config.InterContainerCommunication, []string{"#icc", "-icc"},
        true, "Enable inter-container communication")
        flag.StringVar(&config.GraphDriver, []string{"s", "-storage-driver"}, "",
        "Force the Docker runtime to use a specific storage driver")
        flag.StringVar(&config.ExecDriver, []string{"e", "-exec-driver"}, "native",
        "Force the Docker runtime to use a specific exec driver")
        flag.BoolVar(&config.EnableSelinuxSupport, []string{"-selinux-enabled"},
        false, "Enable selinux support. SELinux does not presently support the
        BTRFS storage driver")
        flag.IntVar(&config.Mtu, []string{"#mtu", "-mtu"}, 0, "Set the containers
        network MTU\nif no value is provided: default to the default route MTU or
        1500 if no default route is available")
        opts.IPVar(&config.DefaultIp, []string{"#ip", "-ip"}, "0.0.0.0", "Default
        IP address to use when binding container ports")
        opts.ListVar(&config.GraphOptions, []string{"-storage-opt"}, "Set storage
        driver options")
        // FIXME: why the inconsistency between "hosts" and "sockets"?
        opts.IPListVar(&config.Dns, []string{"#dns", "-dns"}, "Force Docker to use
        specific DNS servers")
        opts.DnsSearchListVar(&config.DnsSearch, []string{"-dns-search"}, "Force
        Docker to use specific DNS search domains")
}
```

在函数 InstallFlags() 的实现过程中，Docker 主要定义了众多类型不一的 flag 参数，并将该参数的值绑定在 daemonCfg 变量的指定属性上，如：

```
flag.StringVar(&config.Pidfile, []string{"p", "-pidfile"}, "/var/run/docker.pid",
"Path to use for daemon PID file")
```

以上语句的含义为：

- 定义一个 String 类型的 flag 参数。
- 该 flag 的名称为 "p" 或者 "-pidfile"。
- 该 flag 的默认值为 "/var/run/docker.pid"，并将该值绑定在变量 config.Pidfile 上。
- 该 flag 的描述信息为 "Path to use for daemon PID file"。

至此，关于 Docker Daemon 所需要的配置信息均声明并初始化完毕。

3.3.2 flag 参数检查

从本小节开始，程序运行真正进入 Docker Daemon 的 mainDaemon()，下面对此流程进行深入分析。

mainDaemon() 运行的第一个步骤是命令行 flag 参数的检查。具体而言，即当 docker 命令经过 flag 参数解析之后，Docker 判断剩余的参数是否为 0。若为 0，则说明 Docker Daemon 的启动命令无误，正常运行；若不为 0，则说明在启动 Docker Daemon 的时候，传入了多余的参数，此时 Docker 会输出错误提示，并退出运行程序。具体代码如下：

```
if flag.NArg() != 0 {
  flag.Usage()
  return
}
```

3.3.3　创建 engine 对象

在 mainDaemon() 运行过程中，flag 参数检查完毕之后，Docker 随即创建 engine 对象，代码如下：

```
eng := engine.New()
```

Engine 是 Docker 架构中的运行引擎，同时也是 Docker 运行的核心模块。Engine 扮演着 Docker Container 存储仓库的角色，并且通过 Job 的形式管理 Docker 运行中涉及的所有任务。

Engine 结构体的定义位于 ./docker/docker/engine/engine.go#L47-L60，具体代码如下：

```
type Engine struct {
    handlers    map[string]Handler
    catchall    Handler
    hack        Hack // data for temporary hackery (see hack.go)
    id          string
    Stdout      io.Writer
    Stderr      io.Writer
    Stdin       io.Reader
    Logging     bool
    tasks       sync.WaitGroup
    l           sync.RWMutex // lock for shutdown
    shutdown    bool
    onShutdown []func() // shutdown handlers
}
```

Engine 结构体中最为重要的是 handlers 属性，handlers 属性为 map 类型，key 的类型是 string，value 的类型是 Handler。其中 Handler 类型的定义位于 ./docker/docker/engine/engine.go#L23，具体代码如下：

```
type Handler func(*Job) Status
```

可见，Handler 为一个定义的函数。该函数传入的参数为 Job 指针，返回为 Status 状态。

了解完 Engine 以及 Handler 的基本知识之后，我们真正进入创建 Engine 实例的部分，即 New() 函数的实现，具体代码如下：

```
func New() *Engine {
    eng := &Engine{
        handlers: make(map[string]Handler),
        id:       utils.RandomString(),
        Stdout:   os.Stdout,
        Stderr:   os.Stderr,
```

```
        Stdin:   os.Stdin,
        Logging:  true,
    }
    eng.Register("commands", func(job *Job) Status {
        for _, name := range eng.commands() {
            job.Printf("%s\n", name)
        }
        return StatusOK
    })
    // Copy existing global handlers
    for k, v := range globalHandlers {
        eng.handlers[k] = v
    }
    return eng
}
```

分析以上代码，从返回结果可以发现，New() 函数最终返回一个 Engine 实例对象。而在代码实现部分，大致可以将其分为三个步骤：

1）创建一个 Engine 结构体实例 eng，并初始化部分属性，如 handlers、id、标准输出 stdout、日志属性 Logging 等。

2）向 eng 对象注册名为 commands 的 Handler，其中 Handler 为临时定义的函数 func(job *Job) Status{ }，该函数的作用是通过 Job 来打印所有已经注册完毕的 command 名称，最终返回状态 StatusOK。

3）将变量 globalHandlers 中定义完毕的所有 Handler 都复制到 eng 对象的 handlers 属性中。

至此，一个基本的 Engine 对象实例 eng 已经创建完毕，并实现部分属性的初始化。Docker Daemon 启动的后续过程中，仍然会对 Engine 对象实例进行额外的配置。

3.3.4 设置 engine 的信号捕获

Docker 在包 engine 中执行完 Engine 对象的创建与初始化之后，回到 mainDaemon() 函数的运行，紧接着执行的代码为：

```
signal.Trap(eng.Shutdown)
```

Docker Daemon 作为 Linux 操作系统上的一个后台进程，原则上应该具备处理信号的能力。信号处理能力的存在，能保障 Docker 管理员可以通过向 Docker Daemon 发送信号的方式，管理 Docker Daemon 的运行。

再来看以上代码则不难理解其中的含义：在 Docker Daemon 的运行中，设置捕获特定信号后的处理方法，特定信号有 SIGINT、SIGTERM 以及 SIGQUIT；当程序捕获 SIGINT 或者 SIGTERM 信号时，执行相应的善后操作，最后保证 Docker Daemon 程序退出。

该部分代码的实现位于 ./docker/docker/pkg/signal/trap.go。实现的流程分为以下 4 个步骤：

1）创建并设置一个 channel，用于发送信号通知。

2）定义 signals 数组变量，初始值为 os.SIGINT, os.SIGTERM；若环境变量 DEBUG 为空，则添加 os.SIGQUIT 至 signals 数组。

3）通过 gosignal.Notify(c, signals...) 中 Notify 函数来实现将接收到的 signal 信号传递给 c。需要注意的是只有 signals 中被罗列出的信号才会被传递给 c，其余信号会被直接忽略。

4）创建一个 goroutine 来处理具体的 signal 信号，当信号类型为 os.Interrupt 或者 syscall.SIGTERM 时，执行传入 Trap 函数的具体执行方法，形参为 cleanup()，实参为 eng.Shutdown。

Shutdown() 函数的定义位于 ./docker/docker/engine/engine.go#L153-L199，主要完成的任务是：Docker Daemon 关闭时，做一些必要的善后工作。

善后工作主要有以下 4 项：

- Docker Daemon 不再接受任何新的 Job。
- Docker Daemon 等待所有存活的 Job 执行完毕。
- Docker Daemon 调用所有 shutdown 的处理方法。
- 在 15 秒时间内，若所有的 handler 执行完毕，则 Shutdown() 函数返回，否则强制返回。

由于在 signal.Trap(eng.Shutdown) 函数的具体实现中，一旦程序接收到相应的信号，则会执行 eng.Shutdown 这个函数，在执行完 eng.Shutdown 之后，随即执行 os.Exit(0)，完成当前整个 Docker Daemon 程序的退出。源码实现位于 ./docker/docker/pkg/signal/trap.go#L33-L47。

3.3.5　加载 builtins

DockerDaemon 设置完 Trap 特定信号的处理方法（即 eng.shutdown() 函数）之后，Docker Daemon 实现了 builtins 的加载。Docker 的 builtins 可以理解为：Docker Daemon 运行过程中，注册的一些任务（Job），这部分任务一般与容器的运行无关，与 Docker Daemon 的运行时信息有关。加载 builtins 的源码实现如下：

```
if err := builtins.Register(eng); err != nil {
    log.Fatal(err)
}
```

加载 builtins 完成的具体工作是：向 engine 注册多个 Handler，以便后续在执行相应任务时，运行指定的 Handler。这些 Handler 包括：Docker Daemon 宿主机的网络初始化、Web API 服务、事件查询、版本查看、Docker Registry 的验证与搜索等。源码实现位于 ./docker/docker/builtins/builtins.go#L16-L30，如下：

```
func Register(eng *engine.Engine) error {
    if err := daemon(eng); err != nil {
        return err
```

```
        }
        if err := remote(eng); err != nil {
            return err
        }
        if err := events.New().Install(eng); err != nil {
            return err
        }
        if err := eng.Register("version", dockerVersion); err != nil {
            return err
        }
        return registry.NewService().Install(eng)
    }
```

下面分析 Register 函数实现过程中最为主要的 5 个部分：daemon(eng)、remote(eng)、events.New().Install(eng)、eng.Register("version",dockerVersion) 以及 registry.NewService().Install(eng)。

1. 注册网络初始化处理方法

daemon(eng) 的实现过程，主要为 eng 对象注册了一个键为 "init_networkdriver" 的处理方法，此处理方法的值为 bridge.InitDriver 函数，源码如下：

```
func daemon(eng *engine.Engine) error {
    return eng.Register("init_networkdriver", bridge.InitDriver)
}
```

需要注意的是，向 eng 对象注册处理方法，并不代表处理方法的值函数会被立即调用执行，如注册 init_networkdrive 时 bridge.InitDriver 并不会直接运行，而是将 bridge.InitDriver 的函数入口作为 init_networkdriver 的值，写入 eng 的 handlers 属性中。当 Docker Daemon 接收到名为 init_networkdriver 的 Job 的执行请求时，bridge.InitDriver 才被 Docker Daemon 调用执行。

Bridge.InitDriver 的具体实现位于 ./docker/docker/daemon/networkdriver/bridge/driver.go#79-L175，主要作用为：

- 获取为 Docker 服务的网络设备地址。
- 创建指定 IP 地址的网桥。
- 配置网络 iptables 规则。
- 另外还为 eng 对象注册了多个 Handler，如 allocate_interface、release_interface、allocate_port 以及 link 等。

本书将在第 6 章详细分析 Docker Daemon 如何初始化宿主机的网络环境。

2. 注册 API 服务处理方法

remote(eng) 的实现过程，主要为 eng 对象注册了两个 Handler，分别为 serveapi 与 acceptconnections，源码实现如下：

```
func remote(eng *engine.Engine) error {
    if err := eng.Register("serveapi", apiserver.ServeApi); err != nil {
        return err
    }
    return eng.Register("acceptconnections", apiserver.AcceptConnections)
}
```

注册的两个处理方法名称分别为serveapi与acceptconnections，相应的执行方法分别为apiserver.ServeApi与apiserver.AcceptConnections，具体实现位于./docker/docker/api/server/server.go。其中，ServeApi执行时，通过循环多种指定协议，创建出goroutine协调来配置指定的http.Server，最终为不同协议的请求服务；而AcceptConnections的作用主要是：通知宿主机上init守护进程Docker Daemon已经启动完毕，可以让Docker Daemon开始服务API请求。

3. 注册events事件处理方法

events.New().Install(eng)的实现过程，为Docker注册了多个event事件，功能是给Docker用户提供API，使得用户可以通过这些API查看Docker内部的events信息，log信息以及subscribers_count信息。具体的源码位于./docker/docker/events/events.go#L29-L42，如下所示：

```
func (e *Events) Install(eng *engine.Engine) error {
    jobs := map[string]engine.Handler{
        "events":            e.Get,
        "log":               e.Log,
        "subscribers_count": e.SubscribersCount,
    }
    for name, job := range jobs {
        if err := eng.Register(name, job); err != nil {
            return err
        }
    }
    return nil
}
```

4. 注册版本处理方法

eng.Register("version",dockerVersion)的实现过程，向eng对象注册key为version，value为dockerVersion执行方法的Handler。dockerVersion的执行过程中，会向名为version的Job的标准输出中写入Docker的版本、Docker API的版本、git版本、Go语言运行时版本，以及操作系统版本等信息。dockerVersion的源码实现如下：

```
func dockerVersion(job *engine.Job) engine.Status {
    v := &engine.Env{}
    v.SetJson("Version", dockerversion.VERSION)
    v.SetJson("ApiVersion", api.APIVERSION)
    v.Set("GitCommit", dockerversion.GITCOMMIT)
```

```
    v.Set("GoVersion", runtime.Version())
    v.Set("Os", runtime.GOOS)
    v.Set("Arch", runtime.GOARCH)
    if kernelVersion, err := kernel.GetKernelVersion(); err == nil {
        v.Set("KernelVersion", kernelVersion.String())
    }
    if _, err := v.WriteTo(job.Stdout); err != nil {
        return job.Error(err)
    }
    return engine.StatusOK
}
```

5. 注册 registry 处理方法

registry.NewService().Install(eng) 的实现过程位于 ./docker/docker/registry/service.go，功能是：在 eng 对象对外暴露的 API 信息中添加 docker registry 的信息。若 registry.NewService() 被成功安装，则会有两个相应的处理方法注册至 eng，Docker Daemon 通过 Docker Client 提供的认证信息向 registry 发起认证请求；search，在公有 registry 上搜索指定的镜像，目前公有的 registry 只支持 Docker Hub。

Install 的具体实现如下：

```
func (s *Service) Install(eng *engine.Engine) error {
    eng.Register("auth", s.Auth)
    eng.Register("search", s.Search)
    return nil
}
```

至此，Docker Daemon 所有 builtins 的加载全部完成，实现了向 eng 对象注册特定的处理方法。

3.3.6 使用 goroutine 加载 daemon 对象并运行

Docker 执行完 builtins 的加载之后，再次回到 mainDaemon() 的执行流程中。此时，Docker 通过一个 goroutine 协程加载 daemon 对象并开始运行 Docker Server。这一环节的执行，主要包含以下三个步骤：

1）通过 init 函数中初始化的 daemonCfg 与 eng 对象，创建一个 daemon 对象 d。

2）通过 daemon 对象的 Install 函数，向 eng 对象中注册众多的处理方法。

3）在 Docker Daemon 启动完毕之后，运行名为 acceptconnections 的 Job，主要工作为向 init 守护进程发送 READY=1 信号，以便 Docker Server 开始正常接收请求。

源码实现位于 ./docker/docker/docker/daemon.go#L43-L56，如下所示：

```
go func() {
    d, err := daemon.MainDaemon(daemonCfg, eng)
    if err != nil {
        log.Fatal(err)
```

```
    }
    if err := d.Install(eng); err != nil {
        log.Fatal(err)
    }
    if err := eng.Job("acceptconnections").Run(); err != nil {
        log.Fatal(err)
    }
}()
```

下面详细分析三个步骤所做的工作。

1. 创建 daemon 对象

daemon.NewDaemon(daemonCfg, eng) 是创建 daemon 对象 d 的核心部分，主要作用是初始化 Docker Daemon 的基本环境，如处理 config 参数，验证系统支持度，配置 Docker 工作目录，设置与加载多种驱动，创建 graph 环境，验证 DNS 配置等。

由于 daemon.MainDaemon(daemonCfg, eng) 是加载 Docker Daemon 的核心部分，且篇幅过长，本书第 4 章将深入分析 NewDaemon 的实现。

2. 通过 daemon 对象为 engine 注册 Handler

Docker 创建完 daemon 对象，goroutine 立即执行 d.Install(eng)，具体实现位于 ./docker/daemon/daemon.go，代码如下所示：

```
func (daemon *Daemon) Install(eng *engine.Engine) error {
    for name, method := range map[string]engine.Handler{
        "attach":            daemon.ContainerAttach,
        "build":             daemon.CmdBuild,
        "commit":            daemon.ContainerCommit,
        "container_changes":  daemon.ContainerChanges,
        "container_copy":     daemon.ContainerCopy,
        "container_inspect":   daemon.ContainerInspect,
        "containers":         daemon.Containers,
        "create":            daemon.ContainerCreate,
        "delete":            daemon.ContainerDestroy,
        "export":            daemon.ContainerExport,
        "info":              daemon.CmdInfo,
        "kill":               daemon.ContainerKill,
        ...
        "image_delete":      daemon.ImageDelete,
    } {
        if err := eng.Register(name, method); err != nil {
            return err
        }
    }
    if err := daemon.Repositories().Install(eng); err != nil {
        return err
    }
    eng.Hack_SetGlobalVar("httpapi.daemon", daemon)
    return nil
}
```

以上代码的实现同样分为三部分：

- 向 eng 对象中注册众多的处理方法对象。
- daemon.Repositories().Install(eng) 实现了向 eng 对象注册多个与 Docker 镜像相关的 Handler，Install 的实现位于 ./docker/docker/graph/service.go。
- eng.Hack_SetGlobalVar("httpapi.daemon", daemon) 实现向 eng 对象中类型为 map 的 hack 对象中添加一条记录，键为 httpapi.daemon，值为 daemon。

3. 运行名为 acceptconnections 的 Job

Docker 在 goroutine 的最后环节运行名为 acceptconnections 的 Job，主要作用是通知 init 守护进程，使 Docker Daemon 开始接受请求。源码位于 ./docker/docker/docker/daemon.go#L53-L55，如下所示：

```
// after the daemon is done setting up we can tell the api to start
// accepting connections
if err := eng.Job("acceptconnections").Run(); err != nil {
    log.Fatal(err)
}
```

关于 Job 的运行流程大同小异，总结而言，都是首先创建特定名称的 Job，其次为 Job 配置环境参数，最后运行 Job 对应 Handler 的函数。作为本书涉及的第一个具体 Job，下面将对 acceptconnections 这个 Job 的执行进程深入分析。

eng.Job("acceptconnections").Run() 的运行包含两部分：首先执行 eng.Job("acceptconnections")，返回一个 Job 实例，随后再执行该 Job 实例的 Run() 函数。

eng.Job("acceptconnections") 的实现位于 ./docker/docker/engine/engine.go#L115-L137，如下所示：

```
func (eng *Engine) Job(name string, args ...string) *Job {
    job := &Job{
        Eng:    eng,
        Name:   name,
        Args:   args,
        Stdin:  NewInput(),
        Stdout: NewOutput(),
        Stderr: NewOutput(),
        env:    &Env{},
    }
    if eng.Logging {
        job.Stderr.Add(utils.NopWriteCloser(eng.Stderr))
    }
    if handler, exists := eng.handlers[name]; exists {
        job.handler = handler
    } else if eng.catchall != nil && name != "" {
        job.handler = eng.catchall
    }
    return job
}
```

通过分析以上创建 Job 的源码，我们可以发现 Docker 首先创建一个类型为 Job 的 job 对象，该对象中 Eng 属性为函数的调用者 eng，该对象的 Name 属性为 acceptconnections，没有其他参数传入。另外在 eng 对象所有的 handlers 属性中寻找 key 为 acceptconnections 所对应的 value 值（即具体的 Handler）。由于在加载 builtins 时，源码 remote(eng) 已经向 eng 注册过这样一条记录，键为 acceptconnections，值为 apiserver.AcceptConnections。因此，Job 对象的 handler 属性为 apiserver.AcceptConnections。最后函数返回已经初始化完毕的对象 Job。

创建完 Job 对象之后，随即执行该 job 对象的 run() 函数。run() 函数的源码实现位于 ./docker/docker/engine/job.go#L48-L96，该函数执行指定的 Job，并在 Job 执行完成前一直处于阻塞状态。对于名为 acceptconnections 的 Job 对象，运行代码为 job.status = job.handler(job)，由于 job.handler 值为 apiserver.AcceptConnections，故真正执行的是 job.status = apiserver.AcceptConnections(job)。

AcceptConnections 的具体实现属于 Docker Server 的范畴，深入研究 Docker Server 可以发现，这部分源码位于 ./docker/docker/api/server/server.go#L1370-L1380，如下所示：

```
func AcceptConnections(job *engine.Job) engine.Status {
    // Tell the init daemon we are accepting requests
    go systemd.SdNotify("READY=1")
    if activationLock != nil {
        close(activationLock)
    }
    return engine.StatusOK
}
```

AcceptConnections 函数的重点是 go systemd.SdNotify("READY=1") 的实现，位于 ./docker/docker/pkg/system/sdnotify.go#L12-L33，主要作用是通知 init 守护进程 Docker Daemon 的启动已经全部完成，潜在的功能是要求 Docker Daemon 开始接收并服务 Docker Client 发送来的 API 请求。

至此，通过 goroutine 来加载 daemon 对象并运行启动 Docker Server 的工作全部完成。

3.3.7　打印 Docker 版本及驱动信息

Docker 再次回到 mainDaemon() 的运行流程，由于 Go 语言 goroutine 的性质，在 goroutine 执行之时，mainDaemon() 函数内部其他代码也会并发执行。

第一个执行的即为显示 Docker 的版本信息、GitCommit 信息、ExecDriver 和 GraphDriver 这两个驱动的具体信息，源码如下：

```
log.Printf("docker daemon: %s %s; execdriver: %s; graphdriver: %s",
    dockerversion.VERSION,
    dockerversion.GITCOMMIT,
    daemonCfg.ExecDriver,
    daemonCfg.GraphDriver,
)
```

3.3.8 serveapi 的创建与运行

打印 Docker 的部分具体信息之后，Docker Daemon 立即创建并运行名为 serveapi 的 Job，主要作用为让 Docker Daemon 提供 Docker Client 发起的 API 服务。实现代码位于 ./docker/docker/docker/daemon.go#L66，如下所示：

```
job := eng.Job("serveapi", flHosts...)
job.SetenvBool("Logging", true)
job.SetenvBool("EnableCors", *flEnableCors)
job.Setenv("Version", dockerversion.VERSION)
job.Setenv("SocketGroup", *flSocketGroup)

job.SetenvBool("Tls", *flTls)
job.SetenvBool("TlsVerify", *flTlsVerify)
job.Setenv("TlsCa", *flCa)
job.Setenv("TlsCert", *flCert)
job.Setenv("TlsKey", *flKey)
job.SetenvBool("BufferRequests", true)
if err := job.Run(); err != nil {
     log.Fatal(err)
}
```

以上代码标志着 Docker Daemon 真正进入状态。实现过程中，Docker 首先创建一个名为 serveapi 的 Job，并将 flHosts 的值赋给 job.Args。flHosts 的作用主要是：为 Docker Daemon 提供使用的协议与监听的地址。随后，Docker Daemon 为该 Job 设置了众多的环境变量，如安全传输层协议的环境变量等。最后通过 job.Run() 运行该 serveapi 的 Job。

由于在 eng 中 key 为 serveapi 的 handler，value 为 apiserver.ServeApi，故该 Job 运行时，执行 apiserver.ServeApi 函数，位于 ./docker/docker/api/server/server.go。ServeApi 函数的作用是：对于所有用户定义支持协议，Docker Daemon 均创建一个 goroutine 来启动相应的 http.Server，并为每一种协议服务。

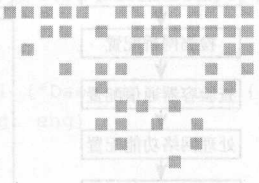
由于创建并启动 http.Server 为 Docker 架构中有关 Docker Server 的重要内容，本书将在第 5 章深入介绍 Docker Server。

至此，我们可以认为 Docker Daemon 已经完成了 serveapi 这个 Job 的初始化工作。一旦 acceptconnections 这个 Job 运行完毕，Docker Daemon 则会通知 init 进程 Docker Daemon 启动完毕，可以开始提供 API 服务，两个 Job 通过 activationLock 进行同步。

3.4 总结

本章从源码的角度分析了 Docker Daemon 的启动，着重分析了 mainDaemon() 的实现。

Docker Daemon 作为 Docker 架构中的主干部分，负责了 Docker 内部几乎所有操作的管理。学习 Docker Daemon 的具体实现，可以对 Docker 架构有一个较为全面的认识。总结而言，Docker 的运行载体为 Daemon，调度管理由 Engine 负责，任务执行靠 Job。



Chapter 4

第 4 章 Docker Daemon 之 NewDaemon 实现

4.1 引言

Docker 的生态系统日趋完善，开发者群体也在日趋庞大，这些现象都使得工业界对 Docker 持续抱有乐观的态度。如今，对于广大开发者而言，使用 Docker 已然不是门槛，享受 Docker 带来的福利也不再困难。然而，如何探寻 Docker 适应的场景，如何发展 Docker 周边的技术，以及如何弥合 Docker 新技术与传统物理机或虚拟机技术之间的鸿沟，已经成为 Docker 研究者们要思考与解决的问题。

在 Docker 架构中 Docker Daemon 支撑着整个后台进程的运行，同时也统一化管理着 Docker 架构中 graph、graphdriver、execdriver、volumes、Docke 容器等众多资源。可以说，Docker Daemon 复杂的运作均由 daemon 对象来调度，而 NewDaemon 的实现恰巧可以帮助大家了解这一切的来龙去脉。通过本章内容的介绍，我们力求帮助广大 Docker 爱好者更多地理解 Docker 的核心——Docker Daemon 的实现。

本章从源码角度，分析 Docker Daemon 加载过程中 NewDaemon 的实现，整个分析过程如图 4-1 所示。

由图 4-1 可见，Docker Daemon 中 NewDaemon 的执行流程主要包含 12 个独立的步骤：处理配置信息、检测系统支持及用户权限、配置工作路径、加载并配置 graphdriver、创建 Docker Daemon 网络环境、创建并初始化 graphdb、创建 execdriver、创建 daemon 实例、检测 DNS 配置、加载已有容器、设置 shutdown 处理方法，以及返回 daemon 实例。

下面我们将在 NewDaemon 的具体实现中，详细分析以上内容。

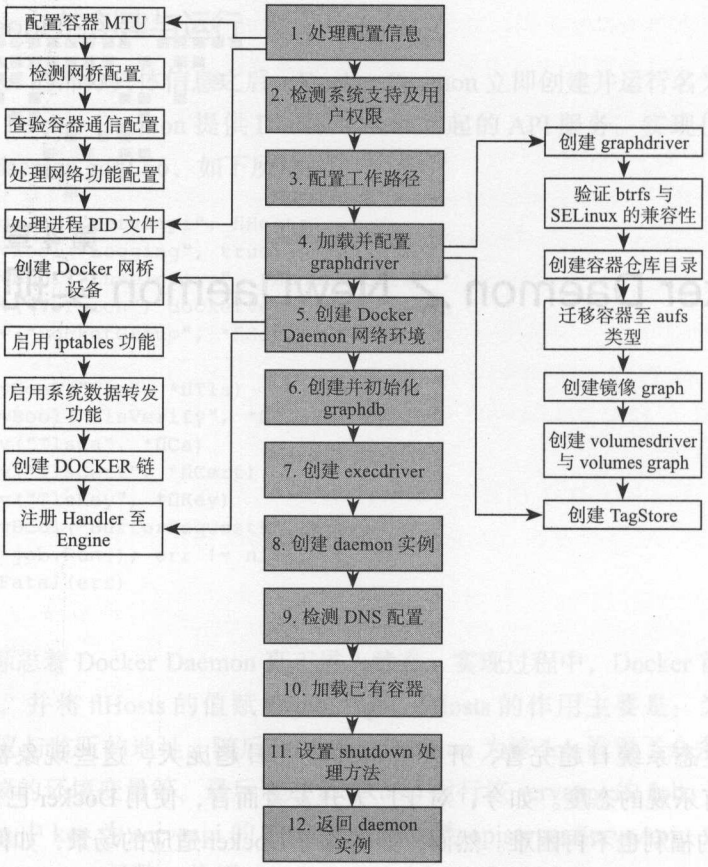


图 4-1 Docker Daemon 中 NewDaemon 执行流程图

4.2 NewDaemon 具体实现

本书第 3 章对于 Docker Daemon 启动的分析过程，有这样一个环节：使用协程加载 daemon 对象。在加载并运行 daemon 对象时，所做的第一个工作是：

```
d, err := daemon.NewDaemon(daemonCfg, eng)
```

简单分析一下这行代码。

- 函数名：NewDaemon
- 调用此函数的包名：daemon
- 函数具体实现源文件：./docker/docker/daemon/daemon.go
- 函数传入实参：daemonCfg，定义 Docker Daemon 运行过程中所需的众多配置信息；eng，在 mainDaemon 中创建的 Engine 对象实例
- 函数返回内容：d，具体的 Daemon 对象实例；err，错误状态

进入 ./docker/docker/daemon/daemon.go 中，寻找函数 NewDaemon 的具体实现，源码如下：

```
func NewDaemon(config *Config, eng *engine.Engine) (*Daemon, error) {
    daemon, err := NewDaemonFromDirectory(config, eng)
    if err != nil {
        return nil, err
    }
    return daemon, nil
}
```

可见，在实现 NewDaemon 的过程中，要通过 NewDaemonFromDirectory 函数来实现创建 Daemon 的运行环境。该函数的实现，传入参数以及返回类型与 NewDaemon 函数完全相同。接下来将详细分析 NewDaemonFromDirectory 的实现细节。

4.3　应用配置信息

NewDaemonFromDirectory 的实现过程中，第一个工作是：如何应用传入的配置信息。这部分配置信息服务于 Docker Daemon 的运行，并在 Docker Daemon 启动初期初始化完毕。配置信息的主要功能是：供用户自由配置 Docker 的可选功能，使得 Docker 的运行更贴近用户期待的运行场景。

配置信息的处理包含 4 部分：

- 配置 Docker 容器的 MTU
- 检测网桥配置信息
- 查验容器间的通信配置
- 处理 PID 文件配置

下面逐一分析配置信息的处理。

4.3.1　配置 Docker 容器的 MTU

config 信息中的 Mtu 属性应用于容器网络接口的最大传输单元（MTU）特性。有关 MTU 的源码如下：

```
if config.Mtu == 0 {
        config.Mtu = GetDefaultNetworkMtu()
}
```

可见，若 config 信息中 Mtu 的值为 0，Docker 则通过 GetDefaultNetworkMtu 函数将 Mtu 设定为默认的值；否则，采用 config 中的 Mtu 值。由于在默认的配置文件 ./docker/docker/daemon/config.go（下文简称为默认配置文件）中，Mtu 属性的初始值为 0，故执行 GetDefaultNetworkMtu。

GetDefaultNetworkMtu 函数的具体实现位于 ./docker/daemon/config.go#L65-L70，如下：

```
func GetDefaultNetworkMtu() int {
    if iface, err := networkdriver.GetDefaultRouteIface(); err == nil {
        return iface.MTU
    }
    return defaultNetworkMtu
}
```

在 GetDefaultNetworkMtu 的实现中，Docker 通过 networkdriver 包的 GetDefaultRouteIface 方法获取具体的网络接口，若该网络接口存在，则返回该网络接口的 MTU 属性值；否则返回默认的 MTU 值 defaultNetworkMtu，值为 1500。

4.3.2 检测网桥配置信息

处理完 config 中的 Mtu 属性之后，Docker 马上检测 config 信息中 BridgeIface 和 BridgeIP 这两个属性。BridgeIface 和 BridgeIP 的作用是为创建网桥的任务 init_networkdriver 提供参数，源码如下：

```
if config.BridgeIface != "" && config.BridgeIP != "" {
    return nil, fmt.Errorf("You specified -b & --bip, mutually exclusive
    options. Please specify only one.")
}
```

以上代码的含义为：若 config 中 BridgeIface 和 BridgeIP 两个属性均不为空，则返回 nil 对象，并返回错误信息，错误信息内容为：用户同时指定了 BridgeIface 和 BridgeIP，这两个属性属于互斥类型，只能指定其中之一。原因是：当用户为 Docker 网桥选定已经存在的网桥接口时，应该沿用已有网桥的当前 IP 地址，不应再提供 IP 地址；当用户不选已经存在的网桥接口作为 Docker 网桥时，Docker 会另行创建一个全新的网桥接口作为 Docker 网桥，此时用户可以为此新创建的网桥接口设定自定义 IP 地址；当然两者都不选的话，Docker 会为用户接管完整的 Docker 网桥创建流程，从创建默认的网桥接口，到尾网桥接口设置默认的 IP 地址。而在默认配置文件中，BridgeIface 和 BridgeIP 的值均为空字符串。

4.3.3 查验容器间的通信配置

查验容器间的通信配置，主要是针对 config 信息中的 EnableIptables 和 Inter-Container Communication 属性。

EnableIptables 属性主要来源于 flag 参数 --iptables，它的作用是：在 DockerDaemon 启动时，是否对宿主机的 iptables 规则作修改。若 EnableIptables 的值为 false，则代表 Docker Daemon 启动时不对宿主机的 iptables 规则作任何修改；若 EnableIptables 的值为 true，则代表 Docker Daemon 启动时对宿主机的 iptables 规则作修改。

仅仅分析 EnableIptables 的作用，显得有些抽象，理解起来也较为生涩。结合

InterContainerCommunication 来分析，就会变得自然很多。InterContainerCommunication 属性来源于 flag 参数 --icc，它的作用是：在 Docker Daemon 启动时，是否开启 Docker 容器之间互相通信的功能。若 InterContainerCommunication 的值为 false，则 Docker Daemon 会在宿主机 iptables 的 FORWARD 链中添加一条 Docker 容器间流量均 DROP 的规则；若 InterContainerCommunication 为 true，则 Docker Daemon 会在宿主机 iptables 的 FORWARD 链中添加一条 Docker 容器间流量均 ACCEPT 的规则。

通过分析以上的内容，我们可以发现：InterContainerCommunication 的值不论为 false 还是为 true，都会在 Docker Daemon 启动时，动用 iptables，故 EnableIptables 的值只能为 true，必须允许 Docker Daemon 启动时对 iptables 规则作修改的操作。另外，若 EnableIptables 为 true，则说明 Docker Daemon 启动时允许对 iptables 规则作修改，而此时若 InterContainerCommunication 为 false，则说明 Docker Daemon 添加一条 DROP 的规则，导致 Docker 容器间不能互相通信，在此情况下，Docker 提供容器间 link 的机制，仍然可以帮助容器实现互相通信。

查验容器间通信配置的源码如下：

```
if !config.EnableIptables && !config.InterContainerCommunication {
    return nil, fmt.Errorf("You specified --iptables=false with --icc=false.
    ICC uses iptables to function. Please set --icc or --iptables to true.")
}
```

代码含义为：若 EnableIptables 和 InterContainerCommunication 两个属性的值均为 false，则返回 nil 对象以及错误信息。其中错误信息为：用户将以上两属性均置为 false，容器间通信需要 iptables 的支持，需设置其中之一为 true。而在默认配置文件中，这两个属性的值均为 true。

4.3.4　处理网络功能配置

接着，Docker 处理 config 中的 DisableNetwork 属性，后续创建并执行创建 Docker Daemon 网络环境时会使用此属性，即在名为 init_networkdriver 的 Job 创建并运行中体现。

```
config.DisableNetwork = config.BridgeIface == DisableNetworkBridge
```

由于 config 信息中的 BridgeIface 属性值为空，另外 DisableNetworkBridge 的值为字符串 none，因此最终 config 中 DisableNetwork 的值为 false。后续名为 init_networkdriver 的 Job 执行时，需要使用 DisableNetwork 这个属性。

4.3.5　处理 PID 文件配置

处理 PID 文件配置，主要工作是：为运行时 Docker Daemon 进程的 PID 号创建一个 PID 文件，文件的路径即为 config 中的 Pidfile 属性，并且为 Docker Daemon 的 shutdown 操作添加一个删除此 Pidfile 的函数，以便在 Docker Daemon 退出的时候，可以在第一时间删除 Pidfile。实现处理 PID 文件配置信息的源码如下：

```
if config.Pidfile != "" {
    if err := utils.CreatePidFile(config.Pidfile); err != nil {
        return nil, err
    }
    eng.OnShutdown(func() {
        utils.RemovePidFile(config.Pidfile)
    })
}
```

在代码执行过程中，首先检测 config 中的 Pidfile 属性是否为空，若为空，则跳过代码块继续执行；若不为空，则首先在文件系统中创建具体的 Pidfile，然后向 eng 的 onShutdown 属性添加一个处理函数，函数具体完成的工作为 utils.RemovePidFile(config.Pidfile)，即在 Docker Daemon 进行 shutdown 操作的时候，删除 Pidfile 文件。在默认配置文件中，Pidfile 文件的初始值为 " /var/run/docker.pid"。

以上便是关于 NewDaemon 实现过程中配置信息的处理分析。

4.4 检测系统支持及用户权限

初步处理完 Docker 的配置信息之后，Docker 立即对自身的运行环境进行一系列的检测。检测主要包括以下三方面：

- 操作系统类型对 Docker Daemon 的支持；
- 用户权限的级别；
- 内核版本与处理器的支持。

系统支持与用户权限检测的实现较为简单，源码实现如下：

```
if runtime.GOOS != "linux" {
    log.Fatalf("The Docker daemon is only supported on linux")
}
if os.Geteuid() != 0 {
    log.Fatalf("The Docker daemon needs to be run as root")
}
if err := checkKernelAndArch(); err != nil {
    log.Fatalf(err.Error())
}
```

首先，通过 runtime.GOOS 检测操作系统的类型。runtime.GOOS 返回运行程序所在操作系统的类型，可以是 Linux、Darwin、FreeBSD 等。结合具体代码，可以发现，若操作系统不为 Linux，将报出 Fatal 错误日志，内容为“Docker Daemon 只能支持 Linux 操作系统”。

接着，通过 os.Geteuid()，检测程序用户是否拥有足够权限。os.Geteuid() 返回调用者所在组的组 id。结合具体源码分析可知，若返回不为 0，则说明 docker 程序不是以 root 用户的身份运行，报出 Fatal 错误日志。

最后，通过 checkKernelAndArch()，检测内核的版本以及主机处理器类型。checkKernel-

AndArch() 的实现同样位于 ./docker/docker/daemon/daemon.go#L1097-L1119。实现过程中，第一个工作是：检测程序运行所在的处理器架构是否为“amd64”，而目前 Docker 运行时只能支持 amd64 的处理器架构。第二个工作是：检测 Linux 内核版本是否满足要求，而目前 Docker Daemon 运行所需的内核版本若过低，则很有可能出现不稳定的状况，因此 Docker 官方建议用户升级内核版本至 3.8.0 或以上版本（包括 3.8.0）。

4.5　配置工作路径

配置 Docker Daemon 的工作路径，主要是创建 Docker Daemon 运行中所在的工作目录。实现过程中，通过 config 中的 Root 属性来完成。Docker Daemon 的 root 目录作用非常大，几乎涵盖 Docker 在宿主机上运行的所有信息，包括：所有的 Docker 镜像内容、所有 Docker 容器的文件系统、所有 Docker 容器的元数据、所有容器的数据卷内容等。

在默认配置文件中，Root 属性的值为”/var/lib/docker”。

在配置工作路径的代码实现中，步骤如下：

1）使用规范路径创建一个 TempDir，路径名为 tmp。

2）通过 tmp，创建一个指向 tmp 的文件符号连接 realTmp。

3）使用 realTemp 的值，创建并赋值给环境变量 TMPDIR。

4）处理 config 的属性 EnableSelinuxSupport。

5）将 realRoot 重新赋值于 config.Root，并创建 Docker Daemon 的工作根目录。

4.6　加载并配置 graphdriver

加载并配置存储驱动 graphdriver，目的在于：使得 Docker Daemon 创建 Docker 镜像管理所需的驱动环境。graphdriver 用于完成 Docker 镜像的管理，包括获取、存储以及容器 rootfs 的构建等。

4.6.1　创建 graphdriver

创建 graphdriver 的内容源码位于 ./docker/docker/daemon/daemon.go#L743-L790。具体细节分析如下：

```
graphdriver.DefaultDriver = config.GraphDriver
driver, err := graphdriver.New(config.Root, config.GraphOptions)
```

首先，Docker 对 graphdriver 包中的 DefaultDriver 对象赋值，值为 config 中的 GraphDriver 属性，在默认配置文件中，GraphDriver 属性的值为空；同样的，属性 GraphOptions 也为空。然后通过 GraphDriver 中的 new 函数实现加载 graph 的存储驱动。

创建具体的 graphdriver 是极其重要的一个环节，实现细节由 graphdriver 包中的 New 函数来完成。New 函数的实现位于 ./docker/docker/daemon/graphdriver/driver.go#L81-L111，实现步骤如下：

第一，遍历数组选择 graphdriver，数组内容为 os.Getenv("DOCKER_DRIVER") 和 DefaultDriver。若数组不为空，graphdriver 则通过 GetDriver 函数直接返回相应的 Driver 对象实例；若为空，则继续往下执行。这部分内容的作用是：让 graphdriver 的加载首先满足用户的自定义选择，用户可以通过定义环境变量的方式定义 graphdriver 的类型，其次 Docker 采用默认值，源码如下：

```
for _, name := range []string{os.Getenv("DOCKER_DRIVER"), DefaultDriver} {
    if name != "" {
        return GetDriver(name, root, options)
    }
}
```

第二，遍历优先级数组 priority 选择 graphdriver，优先级数组 priority 的内容依次为 aufs、btrfs、devicemapper 和 vfs。若遍历验证时，GetDriver 成功，则直接返回当前的 Driver 对象实例；若不成功，则继续往下执行。这部分内容的作用是：在没有指定以及默认的驱动时，从优先级数组中选择驱动，目前优先级最高的为 aufs，源码如下：

```
for _, name := range priority {
    driver, err = GetDriver(name, root, options)
    if err != nil {
        if err == ErrNotSupported || err == ErrPrerequisites || err == ErrIncompatibleFS {
            continue
        }
        return nil, err
    }
    return driver, nil
}
```

第三，从已经注册的 drivers 数组中选择 graphdriver，源码如下：

```
for _, initFunc := range drivers {
    if driver, err = initFunc(root, options); err != nil {
        if err == ErrNotSupported || err == ErrPrerequisites || err ==
        ErrIncompatibleFS {
            continue
        }
            return nil, err
    }
    return driver, nil
}
return nil, fmt.Errorf("No supported storage backend found")
```

在 aufs、btrfs、devicemapper 和 vfs 四个不同类型驱动的 init 函数中，它们均向 graphdriver

的 drivers 数组注册了相应的初始化方法。分别位于 ./docker/docker/daemon/graphdriver/aufs/aufs.go，以及其他三类驱动的相应位置。这部分内容的作用是：在没有优先级数组的时候，同样可以通过注册的驱动来选择具体的 graphdriver。

4.6.2　验证 btrfs 与 SELinux 的兼容性

由于目前在 btrfs 文件系统上运行的 Docker 不兼容 SELinux，因此当 config 中配置信息需要启用 SELinux 的支持并且驱动的类型为 btrfs 时，返回 nil 对象，并报出 Fatal 日志。代码实现如下：

```
// As Docker on btrfs and SELinux are incompatible at present, error on both
being enabled
if config.EnableSelinuxSupport && driver.String() == "btrfs" {
    return nil, fmt.Errorf("SELinux is not supported with the BTRFS graph driver!")
}
```

4.6.3　创建容器仓库目录

Docker Daemon 在创建 Docker 容器之后，需要将容器的元数据信息放置于某个仓库目录下，统一管理。而这个目录即为 daemonRepo，值为："/var/lib/docker/containers"。Docker 通过 daemonRepo 创建对应的目录。源码实现如下：

```
daemonRepo := path.Join(config.Root, "containers")
if err := os.MkdirAll(daemonRepo, 0700); err != nil && !os.IsExist(err) {
    return nil, err
}
```

4.6.4　迁移容器至 aufs 类型

Docker Daemon 的启动很有可能不是第一次，因此 Docker 环境中也有可能存在一部分的遗留内容；Docker Daemon 也有可能是版本升级后的第一次启动，Docker 环境中同样有可能存在遗留内容。因此，当 graphdriver 的类型为 aufs 时，DockerDaemon 启动需要将现有容器 root 目录下的相应内容都迁移至 aufs 类型；若不为 aufs，则继续往下执行。

对于 aufs 类型的 graphdriver，在 Docker 0.7.x 版本之前，Docker 将容器镜像的镜像层内容以及镜像元数据均放在同一个目录下，迁移操作要完成将以上两者拆分存储，以满足新版 Docker 的 aufs 支持。关于 Docker 镜像的存储，可参见第 10 章。

实现源码如下：

```
if err = migrateIfAufs(driver, config.Root); err != nil {
    return nil, err
}
```

这部分的迁移内容主要包括 Repositories、Images 以及 Containers，具体源码实现位于

./docker/docker/daemon/graphdriver/aufs/migrate.go#L39-L50，如下所示：

```
func (a *Driver) Migrate(pth string, setupInit func(p string) error) error {
    if pathExists(path.Join(pth, "graph")) {
        if err := a.migrateRepositories(pth); err != nil {
            return err
        }
        if err := a.migrateImages(path.Join(pth, "graph")); err != nil {
            return err
        }
        return a.migrateContainers(path.Join(pth, "containers"), setupInit)
    }
    return nil
}
```

迁移镜像库的功能是：在 Docker Daemon 的 root 工作目录下创建 repositories-aufs 的文件，存储所有与镜像相关的镜像库以及镜像标签信息。

迁移 Docker 镜像的主要功能是：将原有的 image 镜像都迁移至 aufs 驱动能识别并使用的类型，主要是拆分镜像原先的存储方式，将内容迁移到 aufs 所规定的 layers、diff 与 mnt 目录。

迁移容器的主要功能是：将 Docker 原先的容器运行环境使用 aufs 驱动来进行迁移配置，包括创建容器的 rootfs，配置容器初始层（init layer），创建容器的读写层等。

4.6.5 创建镜像 graph

创建镜像 graph 的主要工作是：通过 Docker 的 root 目录以及 graphdriver 实例，实例化一个全新的 graph 对象，用以管理在文件系统中 Docker 的 root 路径下 graph 目录的内容。graph 目录下的文件以镜像 ID 为单位，分别存储单个镜像的 json 文件以及镜像的大小文件 layersize。其中镜像的 json 文件包含镜像自身的元数据信息。实现源码如下：

```
g, err := graph.NewGraph(path.Join(config.Root, "graph"), driver)
```

NewGraph 的具体实现位于 ./docker/docker/graph/graph.go，实现过程中返回的对象为 Graph 类型，定义如下：

```
type Graph struct {
    Root    string
    idIndex *truncindex.TruncIndex
    driver  graphdriver.Driver
}
```

其中 Root 表示 graph 的工作根目录，一般为 "/var/lib/docker/graph"；idIndex 使得检索字符串标识符时，允许使用任意一个该字符串唯一的前缀，只要该前缀全局唯一，则可确保找到相应的镜像。在这里，idIndex 用于通过简短有效的字符串前缀检索镜像的 ID；最后 driver 表示具体的 graphdriver 类型。

4.6.6　创建 volumesdriver 以及 volumes graph

在 Docker 中数据卷（volume）的概念是：可以从 Docker 宿主机上挂载到 Docker 容器内部的特定目录。一个数据卷可以被多个 Docker 容器挂载，从而使 Docker 容器可以实现互相共享数据等。在实现数据卷时，Docker 需要使用 driver 来管理它，又由于数据卷的管理不会像容器文件系统管理那么复杂，故 Docker 采用 vfs 驱动实现数据卷的管理。

Docker 的范畴中，数据卷可以分为两种：第一种，用户使用 docker run 命令启动容器时传入 -v A:B，使得宿主机上的目录 A 可以挂载到容器内部的目录 B，；第二种，用户在使用 Dockerfile 时使用命令 VOLUME /data，或者使用 dockerrun 命令启动容器时传入 -v /data，用户虽然指定在宿主机上的目录，但是 Docker Daemon 一般情况下会接管宿主机上的目录创建，并再将目录挂载至 Docker 容器内部，此时宿主机上的目录一般位于 "/var/lib/docker/vfs/dir/<ID>"。第一种数据卷通常可以将其称为 bind-mount volume，而第二种通常称为 data volume。

于 volumes graph 上创建 volumesdriver 的源码实现如下：

```
volumesDriver, err := graphdriver.GetDriver("vfs", config.Root, config.
GraphOptions)
volumes, err := graph.NewGraph(path.Join(config.Root, "volumes"), volumesDriver)
```

主要完成工作为：使用 vfs 这种类型的 driver 创建 volumesDriver；在 Docker 的 root 路径下创建 volumes 目录，并返回 volumes 这个 graph 对象实例。

4.6.7　创建 TagStore

TagStore 主要是用于管理存储镜像的仓库列表（repository list）。

创建 tagStore 的源码如下：

```
repositories, err := graph.NewTagStore(path.Join(config.Root, "repositories-
"+driver.String()), g)
```

函数 NewTagStore 的实现位于 ./docker/docker/graph/tags.go，TagStore 的定义如下：

```
type TagStore struct {
    path         string
    graph        *Graph
    Repositories map[string]Repository
    sync.Mutex
    pullingPool map[string]chan struct{}
    pushingPool map[string]chan struct{}
}
```

需要阐述的是 TagStore 类型中的多个属性的含义。

- path：TagStore 中记录镜像仓库的文件所在路径，如 aufs 类型的 TagStore path 的值为 "/var/lib/docker/repositories-aufs"。

- graph：相应的 Graph 实例对象。
- Repositories：记录镜像仓库的映射数据结构。
- sync.Mutex：TagStore 的互斥锁。
- pullingPool：记录池，记录有哪些镜像正在被下载，若某一个镜像正在被下载，则驳回其他 Docker Client 发起下载该镜像的请求。
- pushingPool：记录池，记录有哪些镜像正在被上传，若某一个镜像正在被上传，则驳回其他 Docker Client 发起上传该镜像的请求。

4.7 配置 Docker Daemon 网络环境

创建 Docker Daemon 运行环境的时候，配置 Docker 所在宿主机的网络环境是极为重要的一个环节。这不仅关系着将来容器对外的通信，同样也关系着容器间的通信。

配置 Docker 宿主机的网络环境时，Docker Daemon 通过运行名为 init_networkdriver 的 Job 来完成。源码实现如下：

```
if !config.DisableNetwork {
    job := eng.Job("init_networkdriver")

    job.SetenvBool("EnableIptables", config.EnableIptables)
    job.SetenvBool("InterContainerCommunication", config.InterContainerCommunication)
    job.SetenvBool("EnableIpForward", config.EnableIpForward)
    job.Setenv("BridgeIface", config.BridgeIface)
    job.Setenv("BridgeIP", config.BridgeIP)
    job.Setenv("DefaultBindingIP", config.DefaultIp.String())

    if err := job.Run(); err != nil {
        return nil, err
    }
}
```

分析以上源码可知，通过 config 中的 DisableNetwork 属性来判断是否执行 Job。在默认配置文件中，该属性曾被定义，却没有初始值，然而在应用配置信息这个步骤（4.3 节），Docker 处理网络功能配置时，将 DisableNetwork 属性赋值为 false。故以上判断语句结果为 true，执行相应的代码块。

配置 DockerDaemon 网络环境的工作主要通过 init_networkdriver 这个 Job 来完成。Docker 首先创建名为 init_networkdriver 的 Job，随后为此 Job 设置环境变量，环境变量的值如下：

- 环境变量 EnableIptables，使用 config.EnableIptables 来赋值，默认值为 true。
- 环境变量 InterContainerCommunication，使用 config.InterContainerCommunication 来赋值，为默认值 true。

- 环境变量 EnableIpForward，使用 config.EnableIpForward 来赋值，默认值为 true。
- 环境变量 BridgeIface，使用 config.BridgeIface 来赋值，为空字符串 ""。
- 环境变量 BridgeIP，使用 config.BridgeIP 来赋值，为空字符串 ""。
- 环境变量 DefaultBindingIP，使用 config.DefaultIp.String() 来赋值，默认值为 "0.0.0.0"。

设置完环境变量之后，Docker 随即运行此 Job，由于在 eng 中 key 为 init_networkdriver 的 handler，value 为 bridge.InitDriver 函数，故执行 bridge.InitDriver 函数，具体的实现位于 ./docker/docker/daemon/networkdriver/bridge/dirver.go，作用为：

- 获取为 Docker 容器服务的网络接口 IP 地址。
- 创建指定 IP 地址的网桥接口。
- 启用 Iptables 功能并进行配置。
- 另外，Job 为 eng 实例注册了 4 个 Handler，Handler 名分别为：allocate_interface、release_interface、allocate_port 和 link。

Docker Daemon 的网络初始化关乎 Docker 容器的通信能力，是 Docker 架构中最为基础的知识之一。第 6 章将为大家分析 Docker Daemon 网络环境的创建。

4.7.1　创建 Docker 网络设备

创建 Docker 网络设备，属于 Docker Daemon 创建网络环境的第一步，实际工作是创建名为 docker0 的网桥设备。

在 InitDriver 函数运行过程中，Docker 首先使用 Job 的环境变量初始化内部变量；然后根据目前网络环境，判断是否创建 docker0 网桥，若 Docker 专属网桥已存在，则继续往下执行；否则，创建 docker0 网桥。具体实现为 createBridge(bridgeIP)，以及 createBridgeIface(bridgeIface)。

createBridge 的功能是：在宿主机上启动创建指定名称网桥设备的任务，并为该网桥设备配置一个与其他设备不冲突的网络地址。而 createBridgeIface 通过系统调用负责创建具体实际的网桥设备，并设置 MAC 地址，通过 libcontainer 中 netlink 包的 CreateBridge 来实现。

4.7.2　启用 iptables 功能

创建完网桥之后，Docker Daemon 为未来的 Docker 容器以及宿主机配置 iptables 规则，作用是：为 Docker 容器之间的 link 操作提供 iptables 防火墙支持。源码位于 ./docker/docker/daemon/networkdriver/bridge/driver/driver.go#L133-L137，如下所示：

```
// Configure iptables for link support
if enableIPTables {
    if err := setupIPTables(addr, icc); err != nil {
        return job.Error(err)
    }
}
```

其中 setupIPtables 的调用过程中，addr 地址为 Docker 网桥的网络地址，icc 为 true，即允许 Docker 容器间互相访问。假设网桥设备名为 docker0，网桥网络地址为 docker0_ip, 设置 iptables 规则，具体操作步骤如下：

1）使用 iptables 工具开启新建网桥的 NAT 功能，使用如下命令：

```
iptables -I POSTROUTING -t nat -s docker0_ip ! -o docker0 -j MASQUERADE
```

2）通过 icc 参数，决定是否允许 Docker 容器间的通信，并制定相应 iptables 的 Forward 链。Docker 容器之间建立通信，说明数据包从源容器内发出后，经过 docker0，并且还需要在 docker0 处发往 docker0，最终转向目标容器。换言之，从 docker0 出来的数据包，如果需要继续发往 docker0，则说明是 Docker 容器间的通信数据包。使用命令如下：

```
iptables -I FORWARD -i docker0 -o docker0 -j ACCEPT
```

3）允许接受从容器发出，且目标地址不是容器的数据包。换言之，允许所有从 docker0 发出且不是继续发向 docker0 的数据包，使用命令如下：

```
iptables -I FORWARD -i docker0 ! -o docker0 -j ACCEPT
```

4）对于发往 docker0，并且属于已经建立的连接的数据包，Docker 无条件接受这些连接上的数据包，使用命令如下：

```
iptables -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

4.7.3 启用系统数据包转发功能

在 Linux 系统上，网络设备之间的数据包转发功能是默认禁止的。数据包转发，即当宿主机存在多块网络设备时，如果其中一块网络设备接收到数据包，则无条件将其转发给另外的网络设备。通过修改 /proc/sys/net/ipv4/ip_forward 的值，将其置为 1，则可以保证系统内数据包可以实现转发功能，代码如下：

```
if ipForward {
    // Enable IPv4 forwarding
     if err := ioutil.WriteFile("/proc/sys/net/ipv4/ip_forward", []byte{'1',
     '\n'}, 0644); err != nil {
         job.Logf("WARNING: unable to enable IPv4 forwarding: %s\n", err)
    }
}
```

4.7.4 创建 DOCKER 链

Docker 在网桥设备上创建一条名为 DOCKER 的链，该链的作用是在创建 Docker 容器时实现容器与宿主机的端口映射。实现代码位于 ./docker/docker/daemon/networkdriver/bridge/driver/driver.go，如下所示：

```
if err := iptables.RemoveExistingChain("DOCKER"); err != nil {
    return job.Error(err)
}
if enableIPTables {
    chain, err := iptables.NewChain("DOCKER", bridgeIface)
    if err != nil {
        return job.Error(err)
    }
        portmapper.SetIptablesChain(chain)
}
```

4.7.5　注册处理方法至 Engine

创建完网桥，并配置完基本的 iptables 规则之后，Docker Daemon 在网络方面还向 Engine 中注册了 4 个处理方法，这些处理方法的名称与作用如下：

- allocate_interface：为 Docker 容器分配专属网络接口，分配容器网段的 IP 地址；
- realease_interface：释放 Docker 容器占用的网络接口资源；
- allocate_port：为 Docker 容器分配一个端口；
- link：实现 Docker 容器间的连接操作。

4.8　创建 graphdb 并初始化

graphdb 是一个构建在 SQLite 之上的图形数据库，通常用来记录节点命名以及节点之间的关联。在 Docker 的世界中，用户可以通过 link 操作，使得 Docker 容器之间建立一种关联，而 Docker Daemon 正是使用 graphdb 来记录这种容器间的关联信息。Docker 创建 graphdb 的源码如下：

```
graphdbPath := path.Join(config.Root, "linkgraph.db")
graph, err := graphdb.NewSqliteConn(graphdbPath)
if err != nil {
    return nil, err
}
```

以上代码首先确定 graphdb 的目录为 /var/lib/docker/linkgraph.db；随后通过 graphdb 包内的 NewSqliteConn 打开 graphdb，使用的驱动为 sqlite3，数据源的名称为 /var/lib/docker/linkgraph.db；最后通过 NewDatabase 函数初始化整个 graphdb，为 graphdb 创建 entity 表和 edge 表，并在这两个表中初始化部分数据。NewSqliteConn 函数的实现位于 ./docker/docker/pkg/graphdb/conn_sqlite3.go，代码实现如下：

```
func NewSqliteConn(root string) (*Database, error) {
    ...
    conn, err := sql.Open("sqlite3", root)
    ...
```

```
    return NewDatabase(conn, initDatabase)
}
```

4.9 创建 execdriver

execdriver 是 Docker 中用来执行 Docker 容器任务的驱动。创建并初始化 graphdb 之后，Docker Daemon 随即开始创建了 execdriver，具体源码如下：

```
ed, err := execdrivers.NewDriver(config.ExecDriver, config.Root, sysInitPath, sysInfo)
```

分析以上源码可知，DockerDaemon 创建 execdriver 时，需要以下四部分信息。

- config.ExecDriver：Docker 运行时中用户指定使用的 execdriver 类型，在默认配置文件中值为 native。用户也可以在启动 DockerDaemon 将这个值配置为 lxc，则导致 Docker 使用 lxc 类型的驱动执行 Docker 容器的内部操作。
- config.Root：Docker 运行时的 root 路径，默认配置文件中为 /var/lib/docker。
- sysInitPath：系统中存放 dockerinit 二进制文件的路径，一般为 /var/lib/docker/init/dockerinit-1.2.0。
- sysInfo：系统功能信息，包括：容器的内存限制功能，交换区内存限制功能，数据转发功能，以及 AppArmor 安全功能等。

在执行 execdrivers.NewDriver 之前，首先通过以下代码，获取期望的目标 dockerinit 文件的路径 localPath，以及系统中 dockerinit 文件实际所在的路径 sysInitPath：

```
localCopy := path.Join(config.Root, "init", fmt.Sprintf("dockerinit-%s",
dockerversion.VERSION))
sysInitPath := utils.DockerInitPath(localCopy)
```

通过执行以上代码，localCopy 为 /var/lib/docker/init/dockerinit-1.2.0，而 sysyInitPath 为当前 Docker 运行时中 dockerinit-1.2.0 实际所处的路径，utils.DockerInitPath 实现位于 ./docker/docker/utils/util.go。若 localCopy 与 sysyInitPath 不相等，则说明当前系统中的 dockerinit 二进制文件，不在 localCopy 路径下，则需要将其复制至 localCopy 下，并对该文件设定权限。

设定完 dockerinit 二进制文件的位置之后，Docker Daemon 创建 sysinfo 对象，记录系统的功能属性。SysInfo 的定义位于 ./docker/docker/pkg/sysinfo/sysinfo.go，如下所示：

```
type SysInfo struct {
    MemoryLimit             bool
    SwapLimit               bool
    IPv4ForwardingDisabled  bool
    AppArmor                bool
}
```

其中 Docker Daemon 通过判断 cgroups 文件系统挂载路径下是否均存在 memory.limit_in_

bytes 和 memory.soft_limit_in_bytes 文件来为 MemoryLimit 赋值，若均存在，则置为 true，否则置为 false；通过判断 memory.memsw.limit_in_bytes 文件是否存在来为 SwapLimit 赋值，若该文件存在，则置为 true，否则置为 false。AppArmor 的值则是通过宿主机上是否存在 /sys/kernel/security/apparmor 来判断，若存在，则置为 true，否则置为 false。

执行 execdrivers.NewDriver 时，返回 execdriver.Driver 对象实例，具体代码实现位于 ./docker/docker/daemon/execdriver/execdrivers/execdrivers.go，由于选择使用 native 作为 exec 驱动，故执行以下代码，返回最终的 execdriver，其中 native.NewDriver 实现位于 ./docker/docker/daemon/execdriver/native/driver.go：

```
return native.NewDriver(path.Join(root, "execdriver", "native"), initPath)
```

至此，一个 execdriver 的实例 ed 被 Docker 成功创建。

4.10　创建 daemon 实例

Docker Daemon 在经过以上多个环节的设置之后，整合众多已经创建的对象，创建最终的 Daemon 对象实例 daemon。Daemon 对象实例 daemon 涉及的内容极多，比如：对于 Docker 镜像的存储可以通过 graph 来管理、所有 Docker 容器的元数据信息都保存在 containers 对象中、整个 Docker Daemon 的任务执行位于 eng 属性中，等等。

创建 daemon 实例的源码实现如下：

```
daemon := &Daemon{
    repository:     daemonRepo,
    containers:     &contStore{s: make(map[string]*Container)},
    graph:          g,
    repositories:   repositories,
    idIndex:        truncindex.NewTruncIndex([]string{}),
    sysInfo:        sysInfo,
    volumes:        volumes,
    config:         config,
    containerGraph: graph,
    driver:         driver,
    sysInitPath:    sysInitPath,
    execDriver:     ed,
    eng:            eng,
}
```

分析 Daemon 类型的属性如表 4-1 所示。

表 4-1　Daemon 类型属性分析表

属性名	作用
repository	存储所有 Docker 容器信息的路径，默认为 /var/lib/docker/containers
containers	用于存储 Docker 容器信息的对象

（续）

属性名	作用
graph	存储 Docker 镜像的 graph 对象
repositories	存储本机所有 Docker 镜像 repo 信息的对象
idIndex	用于通过简短有效的字符串前缀定位唯一的镜像
sysInfo	系统功能信息
volumes	管理宿主机上 volumes 内容的 graphdriver，默认为 vfs 类型
config	Config.go 文件中的配置信息，以及执行后产生的配置 DisableNetwork
containerGraph	存放 Docker 镜像关系的 graphdb
driver	管理 Docker 镜像的驱动 graphdriver，默认为 aufs 类型
sysInitPath	系统 dockerinit 二进制文件所在的路径
execDriver	Docker Daemon 的 exec 驱动，默认为 native 类型
eng	Docker 的执行引擎 Engine 类型

4.11 检测 DNS 配置

创建完 Daemon 类型实例 daemon 之后，Docker Daemon 使用 daemon.checkLocaldns() 检测 Docker 运行环境中 DNS 的配置，checkLocaldns 函数的定义位于 ./docker/docker/daemon/daemon.go#L854-L856，代码如下：

```
func (daemon *Daemon) checkLocaldns() error {
    resolvConf, err := resolvconf.Get()
    if err != nil {
        return err
    }
    if len(daemon.config.Dns) == 0 && utils.CheckLocalDns(resolvConf) {
        log.Infof("Local (127.0.0.1) DNS resolver found in resolv.conf and
        containers can't use it. Using default external servers : %v",
        DefaultDns)
        daemon.config.Dns = DefaultDns
    }
    return nil
}
```

以上代码首先通过 resolvconf.Get() 方法获取宿主机 /etc/resolv.conf 中的 DNS 服务器信息。若宿主机上 DNS 文件 resolv.conf 中有 127.0.0.1，而 Docker 容器在自身内部不能使用该地址，故采用默认外在 DNS 服务器，为 8.8.8.8，8.8.4.4；若宿主机上的 resolv.conf 有 Docker 容器可以使用的 DNS 服务器地址，则 Docker Daemon 采用该地址。最终 Docker Daemon 将 DNS 服务器地址赋值给 config 文件中的 Dns 属性。用户通过 Docker Daemon 创建 Docker 容器时，若不指定 DNS 服务器地址，则 Docker Daemon 将会使用 daemon.Config.Dns 作为容器内部的 DNS 服务器地址。

4.12　启动时加载已有 Docker 容器

在 Docker Daemon 重启时，很有可能之前有遗留的 Docker 容器。对于这部分容器，DockerDaemon 重启前，一直将元数据信息存储在 daemon.repository（目录为 /var/lib/docker/containers）中。为了保证重启之后 Docker 容器的信息不丢失，DockerDaemon 首先会进入该目录去查看，是否存在遗留的 Docker 容器。若存在，则 Docker Daemon 加载这部分容器，将容器信息收集，并做相应的维护。

Docker Daemon 加载 Docker 容器的源码实现位于 ./docker/docker/daemon/daemon.go#L854-L856，如下：

```
if err := daemon.restore(); err != nil {
    return nil, err
}
```

需要注意的是：由于 Docker Daemon 的重启不会重启所有重启前运行的容器，故 Docker Daemon 加载已有容器时，会判断容器之前的状态是否为运行，若是的话，会将该容器的状态置为退出，并在内存中的容器对象以及 config.json 文件中将容器主进程的 PID 设为 0。

4.13　设置 shutdown 的处理方法

加载完已有 Docker 容器之后，Docker Daemon 设置了多项在 shutdown 操作中需要执行的处理方法。也就是说，当 Docker Daemon 接收到特定信号，需要执行 shutdown 操作时，先执行这些处理方法完成善后工作，最终再实现物理意义上的 shutdown。实现源码如下：

```
eng.OnShutdown(func() {
    if err := daemon.shutdown(); err != nil {
        log.Errorf("daemon.shutdown(): %s", err)
    }
    if err := portallocator.ReleaseAll(); err != nil {
        log.Errorf("portallocator.ReleaseAll(): %s", err)
    }
    if err := daemon.driver.Cleanup(); err != nil {
        log.Errorf("daemon.driver.Cleanup(): %s", err.Error())
    }
    if err := daemon.containerGraph.Close(); err != nil {
        log.Errorf("daemon.containerGraph.Close(): %s", err.Error())
    }
})
```

由以上源码可见，eng 对象 shutdown 操作执行时，需要执行以上作为参数的 func(){……} 函数。在该函数中，主要完成以下 4 部分操作：

- 运行 daemon 对象的 shutdown 函数，做 daemon 方面的善后工作。
- 通过 portallocator.ReleaseAll()，释放所有之前占用的端口资源。

- 通过 daemon.driver.Cleanup()，通过 graphdriver 实现 unmount 所有有关镜像 layer 的挂载点。
- 通过 daemon.containerGraph.Close() 关闭 graphdb 的连接。

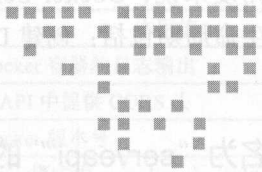
4.14 返回 daemon 对象实例

当所有的工作完成之后，Docker Daemon 返回 daemon 实例，意味着 NewDaemon 函数执行完毕，程序运行最终返回至 mainDaemon() 中，继续通过 goroutine 完成加载 daemon。

4.15 总结

本章从源码的角度深度分析了 Docker Daemon 启动过程中 daemon 对象的创建与加载。在这一环节中涉及内容极多，本章着重归纳总结 daemon 实现的逻辑，一一深入，具体全面。

在 Docker 的架构中，Docker Daemon 的内容是最为丰富全面的，而 NewDaemon 的实现涵盖了 Docker Daemon 启动过程中的大部分工作。可以认为 NewDaemon 是 Docker Daemon 实现过程中的核心所在。深入理解 NewDaemon 的实现，即掌握了 Docker Daemon 运行的来龙去脉。



Chapter 5

第 5 章 Docker Server 的创建

5.1 引言

Docker 架构中，Docker Server 是 Docker Daemon 的重要组成部分。Docker Server 最主要的功能是：接收用户通过 Docker Client 发送的请求，并按照相应的路由规则实现请求的路由分发，最终将请求处理后的结果返回至 Docker Client。

同时，Docker Server 具备十分优秀的用户友好性，多种通信协议的支持大大降低 Docker 用户使用 Docker 的门槛。除此之外，Docker Server 设计实现了详尽清晰的 API 接口，以供 Docker 用户使用。通信安全方面，Docker Server 可以提供安全传输层协议（TLS），保证 Docker Client 与 Docker Server 之间数据的加密传输。并发处理方面，Docker Daemon 大量使用了 Go 语言中的协程 goroutine，大大提高了服务端对于请求的并发处理能力。

本章将从源码的角度分析 Docker Server 的创建，分析内容的安排如下：

1）介绍 Job“serveapi”的创建与执行流程，代表 Docker Server 的创建。

2）深入分析 Job“serveapi”的执行流程。

3）分析 Docker Server 创建 Listener 并服务 API 的流程。

5.2 Docker Server 创建流程

我们在第 3 章主要分析了 Docker Daemon 的启动，而在 mainDaemon() 运行的最后环节，Docker 实现了创建并运行名为 serveapi 的 Job。这一环节的作用是：让 Docker Daemon 提供 API 访问服务。实质上，这正是实现了 Docker 架构中 Docker Server 的创建与运行。

从流程的角度来说，Docker Server 的创建与运行，代表了 Job “serveapi” 的整个生命周期。整个生命周期包括：创建 Docker Server 的 Job，配置 Job 环境变量，以及触发执行 Job。

5.2.1 创建名为 “serveapi” 的 Job

Docker 架构中，Job 是 Engine 内部最基本的任务执行单位。创建 Docker Server，服务于 API 请求，同样属于 Docker 内部的一项任务。因此，这一任务同样需要表示为一个可执行的 Job。换言之，需要创建 Docker Server，则必须创建一个相应的 Job。具体的 Job 创建形式位于 ./docker/docker/docker/daemon.go#L66，如下所示：

```
job := eng.Job("serveapi", flHosts...)
```

以上代码通过 Engine 实例 eng 创建一个 Job 类型的实例 job，Job 实例名为 serveapi，同时用 flHosts 的值初始化 job.Args。flHosts 的作用是：配置 Docker Server 监听的协议与监听的地址。

需要注意的是，第 3 章中函数 mainDaemon() 的具体实现过程中，在加载 builtins 环节已经向 eng 对象注册了键为 serveapi 的处理方法，而该处理方法的值为 api.ServeApi。因此，在运行名为 serveapi 的 Job 时，会执行该 Job 的处理方法 api.ServeApi。

5.2.2 配置 Job 环境变量

创建完 Job 实例 job 之后，Docker Daemon 为 Job 实例配置环境参数。在 Job 的实现过程中，为 Job 配置参数的方式有两种：第一种，创建 Job 实例时，用指定参数直接初始化 Job 的 Args 属性；第二种，创建完 Job 后，给 Job 添加指定的环境变量。以下源码实现了为创建的 job 配置环境变量。

```
job.SetenvBool("Logging", true)
job.SetenvBool("EnableCors", *flEnableCors)
job.Setenv("Version", dockerversion.VERSION)
job.Setenv("SocketGroup", *flSocketGroup)

job.SetenvBool("Tls", *flTls)
job.SetenvBool("TlsVerify", *flTlsVerify)
job.Setenv("TlsCa", *flCa)
job.Setenv("TlsCert", *flCert)
job.Setenv("TlsKey", *flKey)
job.SetenvBool("BufferRequests", true)
```

对于以上配置环境变量的归纳总结如表 5-1 所示。

表 5-1　Job 环境变量列表

环境变量名	flag 参数	默认值	作用值
Logging		true	启用 Docker 容器的日志输出
EnableCors	flEnableCors	false	在远程 API 中提供 CORS 头
Version			显示 Docker 版本号
SocketGroup	flSocketGroup	docker	在 daemon 模式中 unix domain socket 分配用户组名
Tls	flTls	false	使用 TLS 安全传输协议
TlsVerify	flTlsVerify	false	使用 TLS 并验证远程客户端
TlsCa	flCa		指定 CA 文件路径
TlsCert	flCert		TLS 证书文件路径
TlsKey	flKey		TLS 密钥文件路径
BufferRequest		true	缓存 Docker Client 请求

5.2.3　运行 Job

创建完 Job，配置完 Job 的环境变量，意味着 DockerServer 的创建需求已准备完毕。万事俱备，只欠东风，东风就是触发执行这个 Job。Docker 中通过 Job 实例的 run 函数完成 Job 的触发执行。触发执行 serveapi 这个 Job 的具体实现源码如下：

```
if err := job.Run(); err != nil {
    log.Fatal(err)
}
```

由于 Docker 已经在 eng 对象中注册过键为 serveapi 的处理方法，故在运行 job 的时候，执行这个处理方法的值函数，相应处理方法的值为 api.ServeApi。至此，名为 serveapi 的 Job 的生命周期已经完备。本章余下内容将深入分析 Job 的处理方法，api.ServeApi 的执行细节。

5.3　ServeApi 运行流程

ServeApi 属于 Docker Server 提供 API 服务的部分，本小节将从源码的角度剖析 Docker Server 的架构设计与实现。

作为一个监听请求、处理请求、响应请求的服务端，Docker Server 首先需要明确自身可以为多少种通信协议提供服务。稍加深入学习 Docker 这个 C/S 模式的架构设计，就可以发现 Docker Server 支持的协议包括以下三种：TCP 协议、UNIX Socket 形式以及 fd 的形式。随后，Docker Server 根据协议的不同，分别创建不同的服务端实例。最后，在不同的服务端实例中，创建相应的路由模块、监听模块，以及处理请求的处理方法，形成一个完备的服务端。

serveapi 这个 Job 在运行时，将执行 api.ServeApi 函数。ServeApi 的功能是：循环检查 Docker Daemon 当前支持的所有通信协议，并为每一种协议都创建一个协程 goroutine，并在

此协程内部配置一个服务于 HTTP 请求的服务端。ServeApi 的源码实现位于 ./docker/docker/api/server/server.go#L1339，如下所示：

```
func ServeApi(job *engine.Job) engine.Status {
    if len(job.Args) == 0 {
        return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]", job.Name)
    }
    var (
        protoAddrs = job.Args
        chErrors   = make(chan error, len(protoAddrs))
    )
    activationLock = make(chan struct{})

    for _, protoAddr := range protoAddrs {
        protoAddrParts := strings.SplitN(protoAddr, "://", 2)
        if len(protoAddrParts) != 2 {
            return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]", job.Name)
        }
        go func() {
            log.Infof("Listening for HTTP on %s (%s)", protoAddrParts[0],
            protoAddrParts[1])
            chErrors <- ListenAndServe(protoAddrParts[0], protoAddrParts[1], job)
        }()
    }

    for i := 0; i < len(protoAddrs); i += 1 {
        err := <-chErrors
        if err != nil {
            return job.Error(err)
        }
    }
    return engine.StatusOK
}
```

分析以上源码，通过模块化的划分，我们可以发现 ServeApi 的执行流程主要分为以下 4 个步骤：

1）检验 Job 的参数，确保传入参数无误。

2）定义 Docker Server 的监听协议与地址，以及错误信息管道 channel。

3）遍历协议地址，针对协议创建相应的服务端。

4）通过 chErrors 建立 goroutine 与主进程之间的协调关系。

下面详细分析以上 4 个步骤：

第一，Docker Daemon 判断 Job 的参数，保证传入的 Job 参数无误。DockerDaemon 判断的依据来源于 job.Args 的长度。由于 Docker 创建 serveapi 这个 Job 时，通过 flHosts 来初始化 job.Args，故 job.Args 为相当于数组 flHost，若 flHost 的长度为 0，则说明 Docker Server 没有监听的协议与地址，参数有误，返回错误信息。源码如下：

```
if len(job.Args) == 0 {
    return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]", job.Name)
}
```

第二，定义 protoAddrs、chErrors 与 activationLock 三个变量，分别代表 Docker Server 监听的协议与地址，以及 Job 间的同步 channel。

protoAddrs 代表 flHosts 的内容；而 chError 定义了和 protoAddrs 长度一致的错误类型管道，chError 的作用会在下文中说明。同时定义的变量 activationLock，是用以同步 serveapi 和 acceptconnections 这两个 Job 执行的管道。serveapi 运行时，ServeFd 和 ListenAndServe 函数均由于 activationLock 中没有内容而阻塞，而当运行 acceptionconnections 这个 Job 时，该 Job 会首先通知 init 进程 Docker Daemon 已经启动完毕，并关闭 activationLock，因此 ServeFd 以及 ListenAndServe 不再阻塞，结果是 serveapi 继续执行。正是由于 activationLock 的存在，Docker Daemon 可以保证 acceptconnections 这个 Job 的运行有能力通知 serveapi 开启正式服务于 API 请求的功能。源码如下：

```
var (
    protoAddrs = job.Args
    chErrors   = make(chan error, len(protoAddrs))
)
activationLock = make(chan struct{})
```

第三，遍历协议地址，针对协议创建相应的服务端。协议地址即 protoAddrs，也就是 job.Args。DockerDaemon 将 protoAddrs 的每一元素都按照字符串“://”进行分割，若分割后 protoAddrParts 的长度不为 2，则说明协议地址的书写形式有误，返回 Job 错误；若分割后 protoAddrParts 的长度为 2，则说明地址协议符合标准，获取 protoAddrParts 中的协议 protoAddrParts[0] 与地址 protoAddrParts[1]。最后，针对每一次循环中获得的协议与地址，Docker Daemon 均创建一个 goroutine 来执行 ListenAndServe 的操作。goroutine 的运行主要依赖于 ListenAndServe(protoAddrParts[0], protoAddrParts[1], job) 的运行结果。若 ListenAndServe 返回错误，则 chErrors 中有错误，当前协程执行完毕；若没有返回错误，则该协程持续运行，持续提供服务。其中最为重要的是 ListenAndServe 的实现，该函数具体实现了 Docker Daemon 如何创建 listener、router 以及 server，并协调三者进行工作，最终服务于 API 请求。步骤三的源码实现如下：

```
for _, protoAddr := range protoAddrs {
    protoAddrParts := strings.SplitN(protoAddr, "://", 2)
    if len(protoAddrParts) != 2 {
        return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]", job.Name)
    }
    go func() {
        log.Infof("Listening for HTTP on %s (%s)", protoAddrParts[0], protoAddrParts[1])
        chErrors <- ListenAndServe(protoAddrParts[0], protoAddrParts[1], job)
    }()
}
```

第四，根据 chErrors 的值运行，若 chErrors 这个管道中有错误内容，则 ServeApi 的一次循环结束；若无错误内容，则循环被阻塞。chErrors 这个管道的作用是：确保 ListenAndServe 所对应的协程能和主函数 ServeApi 进行协调，如果协程运行出错，主函数 ServeApi 仍然可以捕获这样的错误，从而导致程序的退出。实现源码如下：

```
for i := 0; i < len(protoAddrs); i += 1 {
    err := <-chErrors
    if err != nil {
        return job.Error(err)
    }
}
return engine.StatusOK
```

至此，ServeApi 的运行流程已经全部分析完毕，其中核心部分 ListenAndServe 的实现，将在 5.4 节深入分析。

5.4 ListenAndServe 实现

ListenAndServe 的功能是：使 Docker Server 监听某一指定地址，并接收该地址上的请求，并对以上请求路由转发至相应的处理方法处。从实现的角度来看，ListenAndServe 主要实现了设置一个服务于 HTTP 协议请求的服务端，该服务端监听指定地址上的请求，并对请求做特定的协议检查，最终完成请求的路由与分发。代码实现位于 ./docker/docker/api/server/server.go。

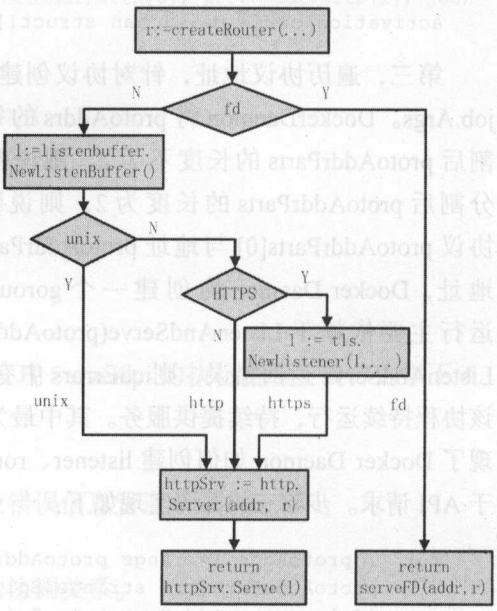


图 5-1 ListenAndServe 执行流程图

ListenAndServe 的实现可以分为以下 4 个部分：

1）创建 router 路由实例。

2）创建 listener 监听实例。

3）创建 http.Server。

4）启动 API 服务。

ListenAndServe 的执行流程如图 5-1 所示。

本节将按照 ListenAndServe 执行流程图一一深入分析各个部分。

5.4.1 创建router路由实例

首先，在函数ListenAndServe的实现过程中，Docker通过createRouter创建了一个router路由实例。代码源码如下：

```
r, err := createRouter(job.Eng, job.GetenvBool("Logging"), job.
GetenvBool("EnableCors"), job.Getenv("Version"))
if err != nil {
    return err
}
```

createRouter的实现位于./docker/docker/api/server/server.go#L1094。

创建router路由实例是一个重要的环节，路由实例的作用是：负责Docker Server对外部请求的路由以及分发。在实现过程中，有两个主要步骤：第一，创建全新的router路由实例；第二，为router实例添加路由记录。

1. 创建空路由实例

实质上，createRouter通过包gorilla/mux来实现一个功能强大的路由器和分发器。源码实现如下：

```
r := mux.NewRouter()
```

NewRouter()函数返回了一个全新的router实例r。在创建Router实例时，给Router对象实例的两个属性进行赋值，分别为nameRoutes和KeepContext。其中namedRoutes属性为map类型，键为string类型，值为Route路由记录类型；另外，KeepContext属性为false，表示Docker Server在处理完请求之后，就清除请求的内容，不对请求做存储操作。源码位于./docker/docker/vendor/src/github.com/gorilla/mux/mux.go#L16，如下所示：

```
func NewRouter() *Router {
    return &Router{namedRoutes: make(map[string]*Route), KeepContext: false}
}
```

可见，以上代码返回的类型为mux.Router。mux.Router会通过一系列已经注册过的路由记录，来匹配接收的请求。首先通过请求的URL或者其他条件，找到相应的路由记录，并调用这条路由记录中的执行处理方法。mux.Router有以下特性：

- 请求可以基于URL的主机名、路径、路径前缀、shemes、请求头和请求值、HTTP请求方法类型或者使用自定义的匹配规则。
- URL主机名和路径可以通过一个正则表达式来表示。
- 注册的URL可以被直接运用，也可以被保留，从而保证维护资源的使用。
- 路由记录同样可以作用于子路由记录：如果父路由记录匹配，则嵌套记录只会被用来测试。当设计一个组内的路由记录共享相同的匹配条件时，如主机名、路径前缀或者其他重复的属性，子路由的方式会起到相应的效果。

❑ mux.Router 实现了 http.Handler 接口，故和标准的 http.ServeMux 兼容。

2. 添加路由记录

Router 路由实例 r 创建完毕，下一步工作是为 Router 实例 r 添加所需要的路由记录。路由记录存储着用来匹配请求的信息，包括对请求的匹配规则，以及匹配之后的处理方法执行入口。

回到 createRouter 实现源码中，Docker 首先判断 Docker Daemon 的启动过程中有没有开启 DEBUG 模式。通过 docker 可执行文件启动 Docker Daemon，解析 flag 参数时，若 flDebug 的值为 false，则说明不需要配置 DEBUG 环境；若 flDebug 的值为 true，则说明需要为 Docker Daemon 添加 DEBUG 功能。具体的源码实现如下：

```
if os.Getenv("DEBUG") != "" {
    AttachProfiler(r)
}
```

AttachProiler(r) 的功能是为路由实例 r 添加与 DEBUG 相关的路由记录，具体实现位于 ./docker/docker/api/server/server.go#L1083，如下所示：

```
func AttachProfiler(router *mux.Router) {
    router.HandleFunc("/debug/vars", expvarHandler)
    router.HandleFunc("/debug/pprof/", pprof.Index)
    router.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
    router.HandleFunc("/debug/pprof/profile", pprof.Profile)
    router.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
    router.HandleFunc("/debug/pprof/heap", pprof.Handler("heap").ServeHTTP)
    router.HandleFunc("/debug/pprof/goroutine", pprof.Handler("goroutine").ServeHTTP)
    router.HandleFunc("/debug/pprof/threadcreate", pprof.Handler
    ("threadcreate").ServeHTTP)
}
```

分析以上源码，可以发现 Docker Server 使用两个包来完成 DEBUG 相关的工作：expvar 和 pprof。包 expvar 为公有变量提供标准化的接口，使得这些公有变量可以通过 HTTP 的形式在“/debug/vars”这个 URL 下被访问，传输格式为 JSON。包 pprof 将 Docker Server 运行时的分析数据通过“/debug/pprof/”这个 URL 向外暴露。这些运行时信息包括以下内容：可得的信息列表、正在运行的命令行信息、CPU 信息、程序函数引用信息、ServeHTTP 函数三部分信息的使用情况（堆使用、协程使用和线程使用）。

再次回到 createRouter 函数实现中，完成 DEBUG 功能的所有工作之后，Docker 创建了一个映射类型的对象 m，用于初始化路由实例 r 的路由记录。简化的 m 对象，源码如下：

```
m := map[string]map[string]HttpApiFunc{
    "GET": {
        "/events":                    getEvents,
        "/info":                      getInfo,
        "/version":                   getVersion,
```

```
        "/images/json":                 getImagesJSON,
        "/images/viz":                  getImagesViz,
        "/images/search":               getImagesSearch,
        "/images/{name:.*}/get":        getImagesGet,
        "/images/{name:.*}/history":    getImagesHistory,
        "/images/{name:.*}/json":       getImagesByName,
        "/containers/ps":               getContainersJSON,
        "/containers/json":             getContainersJSON,
        ...
    },
    "POST": {
        ...
        "/containers/{name:.*}/copy":   postContainersCopy,
    },
    "DELETE": {
        "/containers/{name:.*}": deleteContainers,
        "/images/{name:.*}":     deleteImages,
    },
    "OPTIONS": {
        "": optionsHandler,
    },
}
```

对象 m 的类型为映射，其中键为 string 类型，代表 HTTP 的请求类型，如 GET、POST、DELETE 等，值为另一个映射类型，该映射代表的是 URL 与执行处理方法的映射。在第二个映射类型中，键为 string 类型，代表的是请求 URL，值为 HttpApiFunc 类型，代表具体的执行处理方法。其中 HttpApiFunc 类型的定义如下：

```
type HttpApiFunc func(eng *engine.Engine, version version.Version, w http.
ResponseWriter, r *http.Request, vars map[string]string) error
```

完成对象 m 的定义，随后 Docker Server 通过该对象 m 来添加路由实例 r 的路由记录。对象 m 的请求方法、请求 URL 和请求处理方法这三样内容可以为对象 r 构建一条路由记录。源码实现如下：

```
for method, routes := range m {
    for route, fct := range routes {
        log.Debugf("Registering %s, %s", method, route)
        localRoute := route
        localFct := fct
        localMethod := method

        f := makeHttpHandler(eng, logging, localMethod, localRoute, localFct,
        enableCors, version.Version(dockerVersion))

        if localRoute == "" {
            r.Methods(localMethod).HandlerFunc(f)
        } else {
            r.Path("/v{version:[0-9.]+}" localRoute).Methods(localMethod).
```

```
                HandlerFunc(f)
                r.Path(localRoute).Methods(localMethod).HandlerFunc(f)
            }
        }
    }
```

分析以上源码可以发现：在第一层循环中，按 HTTP 请求方法划分，获得请求方法各自的路由记录；第二层循环，按匹配请求的 URL 进行划分，获得与 URL 相对应的执行处理方法。在嵌套循环中，通过 makeHttpHandler 返回一个执行的函数 f。在返回的这个函数中，涉及了日志配置信息、CORS 信息（跨域资源共享协议），以及版本信息。下面举例说明 makeHttpHandler 的实现，从对象 m 可以看到，对于 GET 请求，请求 URL 为 /info，则请求处理方法为 getInfo。执行 makeHttpHandler 的具体源码实现如下：

```
func makeHttpHandler(eng *engine.Engine, logging bool, localMethod string,
localRoute string, handlerFunc HttpApiFunc, enableCors bool, dockerVersion
version.Version) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // log the request
        log.Debugf("Calling %s %s", localMethod, localRoute)

        if logging {
            log.Infof("%s %s", r.Method, r.RequestURI)
        }

        if strings.Contains(r.Header.Get("User-Agent"), "Docker-Client/") {
            userAgent := strings.Split(r.Header.Get("User-Agent"), "/")
            if len(userAgent) == 2 && !dockerVersion.Equal(version.
            Version(userAgent[1])) {
            log.Debugf("Warning: client and server don't have the same
            version (client: %s, server: %s)", userAgent[1], dockerVersion)
            }
        }
        version := version.Version(mux.Vars(r)["version"])
        if version == "" {
            version = api.APIVERSION
        }
        if enableCors {
            writeCorsHeaders(w, r)
        }

        if version.GreaterThan(api.APIVERSION) {
            http.Error(w, fmt.Errorf("client and server don't have same
            version (client : %s, server: %s)", version, api.APIVERSION).
            Error(), http.StatusNotFound)
            return
        }

        if err := handlerFunc(eng, version, w, r, mux.Vars(r)); err != nil {
```

```
                log.Errorf("Handler for %s %s returned error: %s", localMethod,
                localRoute, err)
                httpError(w, err)
            }
        }
    }
```

可见 makeHttpHandler 的执行直接返回一个函数 func(w http.ResponseWriter, r *http.Request)。在 func 函数的实现中，判断 makeHttpHandler 传入的 logging 参数，若为 true，则将该处理方法的执行日志显示出来；另外通过 makeHttpHandler 传入的 enableCors 参数判断是否在 HTTP 请求的头文件中添加跨域资源共享信息，若为 true，则通过 writeCorsHeaders 函数向请求响应中添加有关 CORS 的 HTTP Header，源码实现位于 ./docker/docker/api/server/server.go#L1022，如下所示：

```
func writeCorsHeaders(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Access-Control-Allow-Origin", "*")
    w.Header().Add("Access-Control-Allow-Headers", "Origin, X-Requested-With,
    Content-Type, Accept")
    w.Header().Add("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT,
    OPTIONS")
}
```

最为重要的执行部分为 handlerFunc(eng, version, w, r, mux.Vars(r))，源码如下：

```
if err := handlerFunc(eng, version, w, r, mux.Vars(r)); err != nil {
    log.Errorf("Handler for %s %s returned error: %s", localMethod,
    localRoute, err)
    httpError(w, err)
}
```

对于 GET 请求类型，URL 为 /info 的请求，由于处理方法名为 getInfo，也就是说 handlerFunc 这个行参的值为 getInfo，故执行部分直接运行 getInfo(eng, version, w, r, mux.Vars(r))，而 getInfo 的具体实现位于 ./docker/docker/api/server/serve.go#L269，如下所示：

```
func getInfo(eng *engine.Engine, version version.Version, w http.ResponseWriter,
r *http.Request, vars map[string]string) error {
    w.Header().Set("Content-Type", "application/json")
    eng.ServeHTTP(w, r)
    return nil
}
```

以上 makeHttpHandler 的执行已经完毕，返回 func 函数，作为指定 URL 对应的处理方法。

创建完处理函数后，Docker 需要向路由实例添加新的路由记录。如果 URL 信息为空，则直接为该 HTTP 请求方法类型添加路由记录；若 URL 不为空，则为请求 URL 路径添加新的路由记录。需要额外注意的是，在 URL 不为空，为路由实例 r 添加路由记录时，考虑

了 API 版本的问题，通过 r.Path("/v{version:[0-9.]+}" + localRoute).Methods(localMethod).HandlerFunc(f) 来实现。

至此，mux.Router 实例 r 的两部分工作已经全部完成：创建空的路由实例 r，为 r 添加相应的路由记录，最后返回配置后的路由实例 r。

5.4.2 创建 listener 监听实例

路由模块，完成请求的路由与分发，是 ListenAndServe 实现中的第一项重要工作。对于请求的监听功能，同样需要模块来完成。而在 ListenAndServe 实现中，第二项重要的工作就是创建 Listener。Listener 是一种面向流协议的通用网络监听模块。

在创建 Listener 之前，Docker 先判断 Docker Server 允许的协议，若协议为 fd 形式，则直接通过 ServeFd 来服务请求；若协议不为 fd 形式，则继续往下执行。

程序首先需要判断 serveapi 这个 Job 的环境中 BufferRequests 的值，若为 true，则通过包 listenbuffer 创建一个 Listener 的实例 l，否则的话直接通过包 net 创建 Listener 实例 l。具体的源码位于 ./docker/docker/api/server/server.go#L1269-L1273，如下所示：

```
if job.GetenvBool("BufferRequests") {
    l, err = listenbuffer.NewListenBuffer(proto, addr, activationLock)
} else {
    l, err = net.Listen(proto, addr)
}
```

由于在 mainDaemon() 中创建 serveapi 这个 Job 之后，给 Job 添加环境变量时，环境变量 BufferRequets 的值为 true，故使用包 listenbuffer 创建 listener 实例。

Listenbuffer 的作用是：让 Docker Server 立即监听指定协议地址上的请求，但是将这些请求暂时先缓存下来，等 Docker Daemon 全部启动完毕之后，才让 Docker Server 开始接受这些请求。这样设计有一个很大的好处，那就是可以保证在 Docker Daemon 还没有完全启动完毕之前，接收并缓存尽可能多的用户请求。

若协议的类型为 TCP，另外 Job 中环境变量 Tls 或者 TlsVerify 有一个为 true，则说明 Docker Server 需要支持 HTTPS 服务，Docker 需要为 Docker Server 配置安全传输层协议（TLS）的支持。实现 TLS 协议，首先需要建立一个 tls.Config 类型实例 tlsConfig，然后在 tlsConfig 中加载证书、认证信息等，最终通过 tls 包中的 NewListener 函数，创建出适应于接收 HTTPS 协议请求的 Listener 实例 l，代码如下：

```
l = tls.NewListener(l, tlsConfig)
```

至此，创建网络监听的 Listener 部分已经全部完成。

5.4.3 创建 http.Server

Docker Server 同样需要创建一个 Server 对象来运行 HTTP/HTTPS 服务端。在

ListenAndServe 实现中第三个重要的工作就是创建 http.Server，实现源码如下：

```
httpSrv := http.Server{Addr: addr, Handler: r}
```

其中 addr 为需要监听的地址，r 为 mux.Router 路由实例。

5.4.4　启动 API 服务

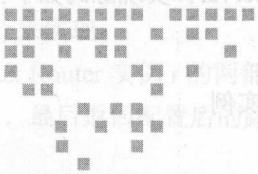
创建 http.Server 实例之后，Docker Server 立即启动 API 服务，使 Docker Server 开始在 Listener 监听实例 l 上接受请求，并对于每一个请求都生成一个新的协程来做专属服务。对于每一个请求，协程会读取请求，查询路由表中的路由记录项，找到匹配的路由记录，最终调用路由记录中的处理方法，执行完毕后，协程对请求返回响应信息。代码如下：

```
return httpSrv.Serve(l)
```

至此，ListenAndServer 的所有流程已经分析完毕，Docker Server 已经开始针对不同的协议，服务 API 请求。

5.5　总结

Docker Server 作为 Docker Daemon 架构中请求的入口，接管了 Docker Client 与 Docker Daemon 之间所有的通信。通信 API 的规范性，通信过程的安全性，服务请求的并发能力，往往都是 Docker 用户最为关心的内容。本章基于 Docker 源码，分析了 Docker Server 大部分的细节实现。希望 Docker 用户可以初探 Docker Server 的设计理念，并且可以更好地利用 Docker Server 创造更大的价值。



Chapter 6

第 6 章
Docker Daemon 网络

6.1 引言

Docker 作为一个开源的轻量级虚拟化容器引擎技术，已然给云计算领域带来全新的发展模式。Docker 借助容器技术彻底释放了轻量级虚拟化技术的威力，让容器的伸缩、应用的运行都变得前所未有的方便与高效。同时，Docker 借助强大的镜像技术，让应用的分发、部署与管理变得史无前例的便捷。然而，Docker 毕竟是一项较为新颖的技术，在 Docker 的世界中，用户并非一劳永逸，其中最为业界所诟病的便是 Docker 的网络问题。

毋庸置疑，对于 Docker 管理者和开发者而言，如何有效、高效地管理 Docker 容器之间的交互以及 Docker 容器的网络一直是一个巨大的挑战。目前，云计算领域中，绝大多数系统都采取分布式技术来设计并实现。然而，在原生态的 Docker 世界中，Docker 的网络却不具备跨宿主机的能力，这也或多或少制约着 Docker 在云计算领域的高速发展。

Docker 网络问题的解决势在必行，面对不同的应用场景，不少 IT 企业都开发了各自的新产品来帮助完善 Docker 的网络。这些企业中不乏像 Google 一样的互联网翘楚，同时也有不少初创企业率先出击，在最前沿不懈探索。这些新产品包括：Google 推出的容器管理和编排工具 Kubernetes，Zett.io 公司开发的通过虚拟网络连接跨宿主机容器的工具 Weave，CoreOS 团队针对 Kubernetes 设计的网络覆盖工具 Flannel，Docker 官方的工程师 Jérôme Petazzoni 自己设计的 SDN 网络解决方案 Pipework，以及 SocketPlane 项目等。

对于 Docker 管理者与开发者而言，虽然 Docker 的跨宿主机通信能力暂时不够完善，但了解 Docker 自身的网络架构也是很有必要的。只有深入了解 Docker 自身的网络设计与实现，才能清楚其弊端，从而扩展 Docker 的跨宿主机能力。

Docker 自身的网络主要包含两部分：Docker Daemon 的网络配置、Docker 容器的网络配置。本章主要从源码的角度，分析 Docker Daemon 在启动过程中，为 Docker 配置的网络环境，内容安排如下：

1）Docker Daemon 网络配置。

2）运行 Docker Daemon 网络初始化任务。

3）创建 Docker 网桥。

6.2　Docker Daemon 网络介绍

在 Docker 环境中，Docker 容器的网络能力一直备受关心。对一个 Docker 容器而言，它可以独享内部的网络栈，并与其他容器分处隔离的网络环境。然而作为 DockerDaemon 创建的容器，容器与宿主机之外建立通信时，仍然无可避免地通过宿主机物理网卡或者虚拟机的 eth0 虚拟网卡。既然如此，那么如何构建 Docker 容器与宿主机之间的网络拓扑，将是一个学习 Docker 网络时必须理解的关键点。

在一台没有安装 Docker 的宿主机上，网络环境很有可能平淡无奇。然而，当用户开始安装并启动 Docker 时，一切发生了变化。而 Docker 管理员完全有权限在启动 Docker 时配置 Docker Daemon 的网络模式。

关于 Docker 的网络模式，大家最熟知的应该就是“桥接”模式，也被称为 bridge 模式。在桥接模式下，Docker 的网络环境拓扑（包括 Docker Daemon 网络环境和 Docker Container 网络环境）如图 6-1 所示。

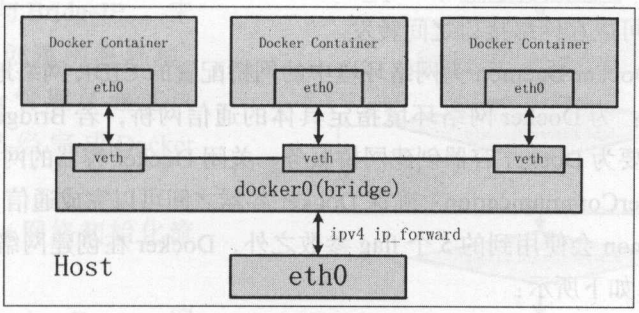


图 6-1　Docker 网络桥接模式示意图

然而，“桥接”模式是 Docker 网络模式中最为常用的模式。除此之外，Docker 还为用户提供了更多的可选项，本章余下部分将进行深入分析。

6.3 Docker Daemon 网络配置接口

Docker Daemon 每次启动的过程中，都会初始化自身的网络环境。初始化后的网络环境最终为 Docker 容器提供网络通信服务。为了实现 Docker Daemon 网络的初始化，Docker 管理员可以在启动 Docker Daemon 时，通过参数的形式配置 Docker 的网络环境。配置参数可以通过运行 docker 二进制可执行文件来完成，即通过运行 docker -d 并添加其他与网络相关的 flag 参数来完成。

其中，涉及的 flag 参数有 EnableIptables、EnableIpForward、BridgeIface、BridgeIP 以及 InterContainerCommunication。这 5 个与网络相关的 flag 参数的定义位于 ./docker/docker/daemon/config.go#L49-L53，具体源码如下：

```
flag.BoolVar(&config.EnableIptables, []string{"#iptables", "-iptables"}, true,
"Enable Docker's addition of iptables rules")
flag.BoolVar(&config.EnableIpForward, []string{"#ip-forward", "-ip-forward"},
true, "Enable net.ipv4.ip_forward")
flag.StringVar(&config.BridgeIP, []string{"#bip", "-bip"}, "", "Use this CIDR
notation address for the network bridge's IP, not compatible with -b")
flag.StringVar(&config.BridgeIface, []string{"b", "-bridge"}, "", "Attach
containers to a pre-existing network bridge\nuse 'none' to disable container
networking")
flag.BoolVar(&config.InterContainerCommunication, []string{"#icc", "-icc"}, true,
"Enable inter-container communication")
```

以下介绍这 5 个 flag 参数的作用：

- EnableIptables：确保 Docker Daemon 启动时，能对宿主机上的 iptables 规则进行修改。
- EnableIpForward：确保 net.ipv4.ip_forward 功能开启，使得宿主机在多网络接口模式下，数据包可以在网络接口之间转发。
- BridgeIP：Docker Daemon 为网络环境中的网桥配置的 CIDR 网络地址。
- BridgeIface：为 Docker 网络环境指定具体的通信网桥，若 BridgeIface 的值为 none，则说明不需要为 Docker 容器创建网桥服务，关闭 Docker 容器的网络能力。
- InterContainerCommunication：确保 Docker 容器之间可以完成通信，通过防火墙完成。

除 DockerDaemon 会使用到的 5 个 flag 参数之外，Docker 在创建网络环境时，还使用一个 DefaultIP 变量，如下所示：

```
opts.IPVar(&config.DefaultIp, []string{"#ip", "-ip"}, "0.0.0.0", "Default IP
address to use when binding container ports")
```

该变量的作用是：绑定 Docker 容器的端口与宿主机上的某一个端口时，将 DefaultIp 作为默认使用的宿主机 IP 地址。

具备以上 Docker Daemon 的网络背景知识之后，我们来分析如何通过 docker 二进制文件启动 Docker Dameon 并配置相应的网络环境。Docker Daemon 的网络环境和两个 flag 参数相关性最大，分别为 BridgeIP 和 BridgeIface。举例说明如表 6-1 所示。

表 6-1　Docker Daemon 启动命令表

启动 Docker Daemon 命令	作用分析
docker -d	启动 Docker Daemon，使用默认网桥 docker0，不指定 CIDR 网络地址
docker -d -b="xxx"	启动 Docker Daemon，使用网桥 xxx，不指定 CIDR 网络地址
docker -d --bip="172.17.42.1"	启动 Docker Daemon，使用默认网桥 docker0，使用指定 CIDR 网络地址 172.17.42.1
docker -d --bridge="xxx" --bip="10.0.42.1"	报错，出现兼容性问题，不能同时指定 BridgeIP 和 BridgeIface
docker -d --bridge="none"	启动 Docker Daemon，不创建 Docker 网络环境

深入理解 BridgeIface 与 BridgeIP，并熟练使用相应的 flag 参数，就能做到配置 Docker Daemon 的网络环境。需要特别注意的是，Docker Daemon 的网络与 Docker 容器的网络存在很大的区别。Docker Daemon 为 Docker 容器创建网络的大环境，Docker 容器的网络需要 Docker Daemon 的网络提供支持，但不唯一。举一个形象的例子，Docker Daemon 可以创建 docker0 网桥，为之后 Docker 容器的桥接模式提供支持；然而在桥接模式下，Docker 容器可以启用桥接，转而根据用户需求创建自身网络。其中 Docker 容器的网络可以是桥接模式的网络，同时也可以直接共享宿主机的网络设备，另外还有其他模式。关于 Docker 容器的网络，将在第 7 章进行详细介绍。

6.4　Docker Daemon 网络初始化

正如上一节所言，Docker 管理员可以通过与网络相关的 flag 参数 BridgeIface 与 BridgeIP，来为 Docker Daemon 创建网络环境。最简单的，Docker 管理员通过执行 "docker -d" 就已经完成 Docker Daemon 的运行。

Docker Daemon 网络初始化流程如图 6-2 所示。

总体而言，Docker Daemon 网络的初始化流程主要是根据解析 flag 参数来决定到底建立哪种类型的网络环境。从流程图中可知，Docker Daemon 创建网络环境时有两个分支，不难发现分支代表的分别是：为 Docker 创建一个网络驱

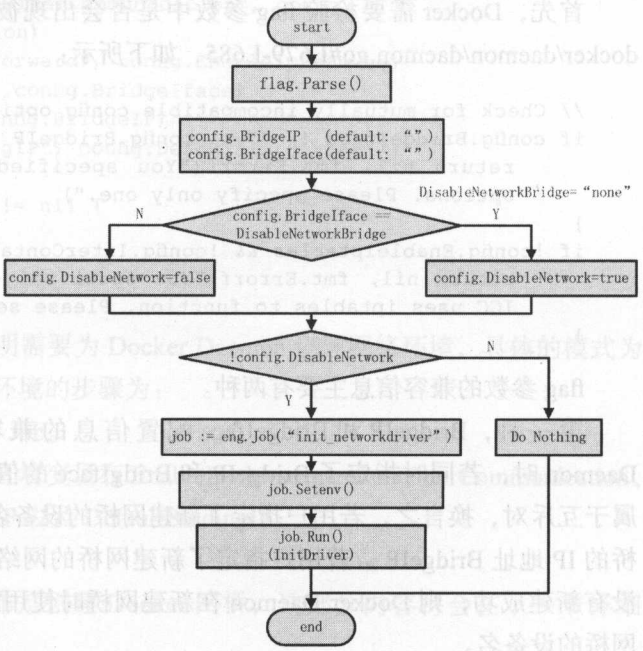


图 6-2　Docker Daemon 网络初始化流程图

动，以及对 Docker 的网络不做任何操作。

下面参照 Docker Daemon 网络初始化流程图具体分析实现步骤。

6.4.1 启动 Docker Daemon 传递 flag 参数

用户希望感受到 Docker 带来的便利，首先必须要有一个运行着的 Docker Daemon。因此，用户第一步要完成的就是启动 Docker Daemon。用户可以通过 docker 可执行文件来启动 Docker Daemon，并在命令行中选择性地传入所需要的 flag 参数。

6.4.2 解析网络 flag 参数

Docker Daemon 的启动前期，Docker 使用 flag 包对命令行中的 flag 参数进行解析。其中和 Docker Daemon 网络配置相关的 flag 参数有五个，分别是：EnableIptables、EnableIpForward、BridgeIP、BridgeIface 以及 InterContanierCommunication，各个 flag 参数的作用本章已有介绍。

6.4.3 预处理 flag 参数

解析完与 Docker Daemon 网络相关的 flag 参数之后，Docker 仍然需要对这些参数的值进行预处理。预处理与网络配置相关的 flag 参数信息，包括检测配置信息的兼容性，以及判断是否创建 Docker 网络环境。

首先，Docker 需要检验 flag 参数中是否会出现彼此不兼容的信息，源码位于 ./docker/docker/daemon/daemon.go#L679-L685，如下所示：

```
// Check for mutually incompatible config options
if config.BridgeIface != "" && config.BridgeIP != "" {
    return nil, fmt.Errorf("You specified -b & --bip, mutually exclusive
    options. Please specify only one.")
}
if !config.EnableIptables && !config.InterContainerCommunication {
    return nil, fmt.Errorf("You specified --iptables=false with --icc=false.
    ICC uses iptables to function. Please set --icc or --iptables to true.")
}
```

flag 参数的兼容信息主要有两种。

第一种，BridgeIP 和 BridgeIface 配置信息的兼容性。具体表现为用户启动 Docker Daemon 时，若同时指定了 BridgeIP 和 BridgIface 的值，则出现兼容问题。原因在于这两者属于互斥对，换言之，若用户指定了新建网桥的设备名，那么该网桥已经存在，无须指定网桥的 IP 地址 BridgeIP；若用户指定了新建网桥的网络 IP 地址 BridgeIP，那么该网桥肯定还没有新建成功，则 Docker Daemon 在新建网桥时使用默认网桥名 docker0，无须在另行指定网桥的设备名。

第二种，EnableIptables 和 InterContainerCommunication 配置信息的兼容性。具体表现为

不能同时指定这两个 flag 参数为 false。原因很简单，若指定 InterContainerCommunication 为 false，则说明 Docker Daemon 不允许创建的 Docker 容器之间进行互相通信。但是为了达到以上目的，Docker 正是使用 iptables 防火墙的过滤规则。因此，再次设定 EnableIptables 为 false，关闭 iptables 的使用，即出现了自相矛盾的结果。

检验完系统配置信息的兼容性问题，Docker Daemon 接着会判断是否需要为 Docker Daemon 配置网络环境。判断的依据为 BridgeIface 的值是否与 DisableNetworkBridge 的值相等，DisableNetworkBridge 在 ./docker/docker/daemon/config.go#L13 中被定义为 const 常量，值为字符串 none。因此，若 BridgeIface 为 none，则 DisableNetwork 为 true，最终 Docker Daemon 不会创建网络环境；若 BridgeIface 不为 none，则 DisableNetwork 为 false，最终 Docker Daemon 需要创建网络环境（桥接模式）。

6.4.4　确定 Docker 网络模式

Docker 网络模式由配置信息 DisableNetwork 决定。由于在上一节 Docker Daemon 已经得出 DisableNetwork 的值，故本节可以确定 Docker 网络模式。这部分的源码实现位于 ./docker/docker/daemon/daemon.go#L792-L805，如下所示：

```
if !config.DisableNetwork {
    job := eng.Job("init_networkdriver")

    job.SetenvBool("EnableIptables", config.EnableIptables)
    job.SetenvBool("InterContainerCommunication", config.
    InterContainerCommunication)
    job.SetenvBool("EnableIpForward", config.EnableIpForward)
    job.Setenv("BridgeIface", config.BridgeIface)
    job.Setenv("BridgeIP", config.BridgeIP)
    job.Setenv("DefaultBindingIP", config.DefaultIp.String())

    if err := job.Run(); err != nil {
        return nil, err
    }
}
```

若 DisableNetwork 为 false，则说明需要为 Docker Daemon 创建网络环境，具体的模式为网桥模式。创建 Docker Daemon 网络环境的步骤为：

1）创建名为 init_networkdriver 的 Job。

2）为 Job 配置环境变量，配置的环境变量有 EnableIptables、InterContainerCommunication、EnableIpForward、BridgeIface、BridgeIP 以及 DefaultBindingIP。

3）触发执行 Job。

运行 init_network 实际完成的工作是创建 Docker 网桥，这部分内容将会在下一节详细分析。

若 DisableNetwork 为 true。则说明不需要为 Docker Daemon 创建网络环境，网络模式属

于 none 模式。

以上便是 Docker Daemon 网络初始化的所有流程。

6.5 创建 Docker 网桥

Docker 的网络往往是 Docker 开发者最常提起的话题。而 Docker 网络中最常使用的模式为桥接模式。本节将详细分析 Docker 网桥的创建流程。

Docker 网桥的创建通过 init_network 这个 Job 的运行来完成。init_network 的实现为 InitDriver 函数，位于./docker/docker/daemon/networkdriver/bridge/driver.go#L79，InitDriver 函数的运行流程如图 6-3 所示。

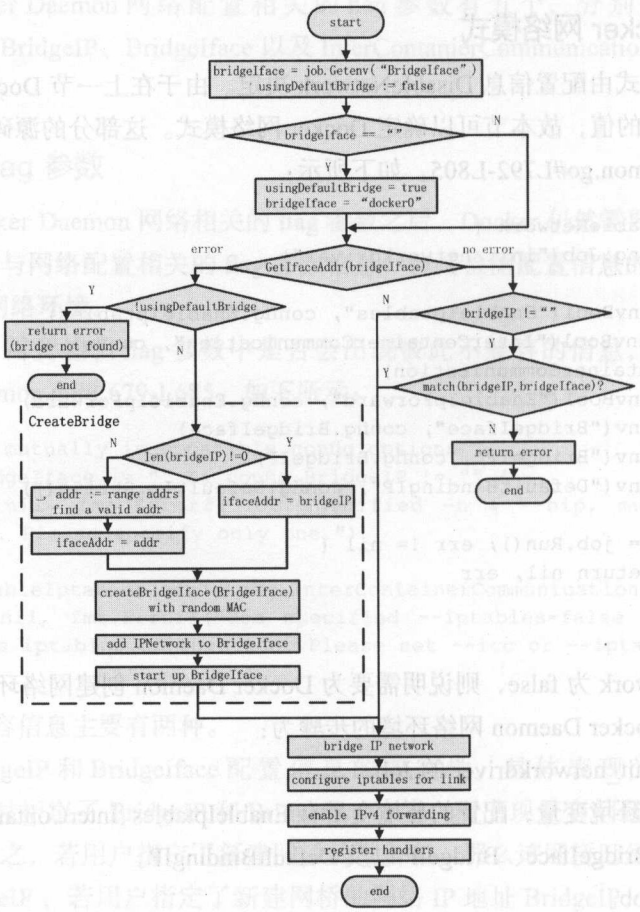


图 6-3 Docker Daemon 创建网桥流程图

6.5.1　提取环境变量

在 InitDriver 函数的实现过程中，Docker 首先提取 init_networkdriver 这个 Job 的环境变量。这样的环境变量共有 6 个，各自的作用在上文已经详细说明。具体的实现源码如下：

```
var (
    network        *net.IPNet
    enableIPTables = job.GetenvBool("EnableIptables")
    icc            = job.GetenvBool("InterContainerCommunication")
    ipForward      = job.GetenvBool("EnableIpForward")
    bridgeIP       = job.Getenv("BridgeIP")
)

if defaultIP := job.Getenv("DefaultBindingIP"); defaultIP != "" {
    defaultBindingIP = net.ParseIP(defaultIP)
}

bridgeIface = job.Getenv("BridgeIface")
```

6.5.2　确定 Docker 网桥设备名

提取 Job 的环境变量之后，Docker 随即确定最终使用网桥设备的名称。为此，Docker 首先创建了一个名为 usingDefaultBridge 的 bool 变量，含义为是否使用默认的网桥设备，默认值为 false。接着，若环境变量中 bridgeIface 的值为空，则说明用户启动 Docker 时，没有指定特定的网桥设备名，因此 Docker 首先将 usingDefaultBridge 置为 true，然后使用默认的网桥设备名 DefaultNetworkBridge 赋值于 bridgeIface，即 docker0；若 bridgeIface 的值不为空，则判断条件不成立，继续往下执行。这部分的源码实现为：

```
usingDefaultBridge := false
if bridgeIface == "" {
    usingDefaultBridge = true
    bridgeIface = DefaultNetworkBridge
}
```

6.5.3　查找 bridgeIface 网桥设备

确定 Docker 网桥设备名 bridgeIface 之后，Docker 首先通过 bridgeIface 设备名在宿主机上查找该设备是否真实存在。若存在，则返回该网桥设备的 IP 地址，若不存在，则返回 nil。实现源码位于 ./docker/docker/daemon/networkdriver/bridge/driver.go#L99，如下所示：

```
addr, err := networkdriver.GetIfaceAddr(bridgeIface)
```

GetIfaceAddr 的实现位于 ./docker/docker/daemon/networkdriver/utils.go，实现步骤为：首先通过 Golang 中 net 包的 InterfaceByName 方法获取名为 bridgeIface 的网桥设备，会得出以下结果：

- 若名为 bridgeIface 的网桥设备不存在，直接返回错误。
- 若名为 bridgeIface 的网桥设备存在，返回该网桥设备的 IP 地址。

需要强调的是：GetIfaceAddr 函数返回错误，说明当前宿主机上不存在名为 bridgeIface 的网桥设备。而这样的结果会有两种不同的情况：第一，用户指定了 bridgeIface，那么 usingDefaultBridge 为 false，而该 bridgeIface 网桥设备在宿主机上不存在；第二，用户没有指定 bridgeIface，那么 usingDefaultBridge 为 true，bridgeIface 名为 docker0，而 docker0 网桥在宿主机上也不存在。

当然，若 GetIfaceAddr 函数返回的是一个 IP 地址，则说明当前宿主机上存在名为 bridgeIface 的网桥设备。这样的结果同样会有两种不同的情况：第一，用户指定了 bridgeIface，那么 usingDefaultBridge 为 false，而该 bridgeIface 网桥设备在宿主机上已经存在；第二，用户没有指定 bridgeIface，那么 usingDefaultBridge 为 true，bridgeIface 名为 docker0，而 docker0 网桥在宿主机上也已经存在。第二种情况一般是：用户在宿主机上第一次启动 Docker Daemon 时，创建了默认网桥设备 docker0，而后 docker0 网桥设备一直存在于宿主机上，故之后在不指定网桥设备的情况下，重启 Docker Daemon，会出现 docker0 已经存在的情况。

以下两节将分别从 bridgeIface 已创建与 bridgeIface 未创建两种不同的情况进行深入分析。

6.5.4 bridgeIface 已创建

Docker Daemon 所在宿主机上 bridgeIface 的网桥设备存在时，Docker Daemon 仍然需要验证用户在配置信息中是否为网桥设备指定了 IP 地址。

用户启动 Docker Daemon 时，假如没有指定 bridgeIP 参数信息，则 Docker Daemon 使用名为 bridgeIface 的原有的 IP 地址。

当用户指定了 bridgeIP 参数信息时，则需要验证：指定的 bridgeIP 参数信息与 bridgeIface 网桥设备原有的 IP 地址信息是否匹配。若两者匹配，则验证通过，继续往下执行；若两者不匹配，则验证不通过，抛出错误，显示“bridgeIP 与已有网桥配置信息不匹配”。该部分内容位于 ./docker/docker/daemon/networkdriver/bridge/driver.go#L119-L129，源码如下：

```
network = addr.(*net.IPNet)
// validate that the bridge ip matches the ip specified by BridgeIP
if bridgeIP != "" {
    bip, _, err := net.ParseCIDR(bridgeIP)
    if err != nil {
        return job.Error(err)
    }
    if !network.IP.Equal(bip) {
        return job.Errorf("bridge ip (%s) does not match existing bridge
        configuration %s", network.IP, bip)
    }
}
```

6.5.5　bridgeIface 未创建

Docker Daemon 所在宿主机上 bridgeIface 的网桥设备未创建时，上文已经介绍将存在两种情况：

- 用户指定的 bridgeIface 未创建。
- 用户未指定 bridgeIface，而 docker0 暂未创建。

当用户指定的 bridgeIface 不存在于宿主机时，即没有使用 Docker 的默认网桥设备名 docker0，Docker 打印日志信息“指定网桥设备未找到”，并返回网桥未找到的错误信息。源码实现如下：

```
if !usingDefaultBridge {
    job.Logf("bridge not found: %s", bridgeIface)
    return job.Error(err)
}
```

当使用默认网桥设备名，而 docker0 网桥设备还未创建时，Docker Daemon 则立即实现创建网桥的操作，并返回该 docker0 网桥设备的 IP 地址。代码如下：

```
// If the iface is not found, try to create it
job.Logf("creating new bridge for %s", bridgeIface)
if err := createBridge(bridgeIP); err != nil {
    return job.Error(err)
}

job.Logf("getting iface addr")
addr, err = networkdriver.GetIfaceAddr(bridgeIface)
if err != nil {
    return job.Error(err)
}
network = addr.(*net.IPNet)
```

创建 Docker Daemon 网桥设备 docker0 的实现，全部由 createBridge(bridgeIP) 来实现，createBridge 的实现位于 ./docker/docker/daemon/networkdriver/bridge/driver.go#L245。createBridge 函数实现步骤如下：

1）确定网桥设备 docker0 的 IP 地址。

2）通过 createBridgeIface 函数创建 docker0 网桥设备，并为网桥设备分配随机的 MAC 地址。

3）将第一步中已经确定的 IP 地址，添加给新创建的 docker0 网桥设备。

4）启动 docker0 网桥设备。

下面详细分析 4 个步骤的具体实现。

第一步，Docker Daemon 确定 docker0 的 IP 地址，实现方式为判断用户是否为网桥设备指定 bridgeIP。若用户未指定 bridgeIP，则从 Docker 预先准备的 IP 网段列表 addrs 中查找合适的网段。具体的源码实现位于 ./docker/docker/daemon/networkdriver/bridge/driver.

go#L257-L278，如下所示：

```
if len(bridgeIP) != 0 {
        _, _, err := net.ParseCIDR(bridgeIP)
        if err != nil {
                return err
        }
        ifaceAddr = bridgeIP
} else {
        for _, addr := range addrs {
                _, dockerNetwork, err := net.ParseCIDR(addr)
                if err != nil {
                        return err
                }
                if err := networkdriver.CheckNameserverOverlaps(nameservers,
                dockerNetwork); err == nil {
                        if err := networkdriver.CheckRouteOverlaps(dockerNetwork); err == nil {
                                ifaceAddr = addr
                                break
                        } else {
                                log.Debugf("%s %s", addr, err)
                        }
                }
        }
}
```

其中，Docker Daemon 为网桥设备准备的候选网段地址 addrs 为：

```
addrs = []string{
        "172.17.42.1/16", // Don't use 172.16.0.0/16, it conflicts with EC2 DNS
        172.16.0.23
        "10.0.42.1/16", // Don't even try using the entire /8, that's too intrusive
        "10.1.42.1/16",
        "10.42.42.1/16",
        "172.16.42.1/24",
        "172.16.43.1/24",
        "172.16.44.1/24",
        "10.0.42.1/24",
        "10.0.43.1/24",
        "192.168.42.1/24",
        "192.168.43.1/24",
        "192.168.44.1/24",
}
```

通过执行以上流程，Docker Daemon 可以确定一个可用的 IP 网段地址，为 ifaceAddr；若仍然未匹配合适的网段地址，DockerDaemon 返回错误日志，表明没有合适的 IP 地址赋予 docker0 网桥设备。

第二步，DockerDaemon 通过 createBridgeIface 函数创建 docker0 网桥设备。createBridgeIface 函数的源码实现如下：

```
func createBridgeIface(name string) error {
    kv, err := kernel.GetKernelVersion()
    // only set the bridge's mac address if the kernel version is > 3.3
    // before that it was not supported
    setBridgeMacAddr := err == nil && (kv.Kernel >= 3 && kv.Major >= 3)
    log.Debugf("setting bridge mac address = %v", setBridgeMacAddr)
    return netlink.CreateBridge(name, setBridgeMacAddr)
}
```

以上代码通过宿主机Linux内核信息，确定宿主机内核版本是否支持设定网桥设备的MAC地址。若Linux内核版本大于3.3，则支持配置MAC地址，否则不支持。而Docker在不低于3.8的内核版本上才能稳定运行，故可以认为内核支持配置MAC地址。最后通过netlink的CreateBridge函数实现创建docker0网桥。

Netlink是Linux中一种较为特殊的socket通信方式，提供了用户应用间和内核进行双向数据传输的途径。在这种模式下，用户态可以使用标准的socket API来使用netlink强大的功能，而内核态需要使用专门的内核API才能使用netlink。

libcontainer的netlink包中的CreateBridge实现了创建实际的网桥设备，具体使用系统调用的代码如下：

```
syscall.Syscall(syscall.SYS_IOCTL, uintptr(s), SIOC_BRADDBR, uintptr(unsafe.
Pointer(nameBytePtr)))
```

创建完网桥设备之后，为docker0网桥设备配置MAC地址，实现函数为setBridge-MacAddress。

第三步为创建的docker0网桥设备绑定IP地址。上一步仅完成了创建名为docker0的网桥设备，之后仍需要为docker0网桥设备绑定IP地址。具体源码实现为：

```
if netlink.NetworkLinkAddIp(iface, ipAddr, ipNet); err != nil {
    return fmt.Errorf("Unable to add private network: %s", err)
}
```

NetworkLinkAddIP的实现同样位于libcontainer中的netlink包，主要的功能为：通过netlink机制为一个网络接口设备绑定一个IP地址。

第四步是启动docker0网桥设备。具体实现代码如下：

```
if err := netlink.NetworkLinkUp(iface); err != nil {
    return fmt.Errorf("Unable to start network bridge: %s", err)
}
```

NetworkLinkUp的实现同样位于libcontainer中的netlink包，功能为启动docker0网桥设备。

至此，docker0网桥历经确定IP、创建、绑定IP、启动四个环节，createBridge关于docker0网桥设备的工作全部完成。

6.5.6 获取网桥设备的网络地址

创建完网桥设备之后，网桥设备必然会存在一个网络地址。网桥网络地址的作用为：Docker Daemon 在创建 Docker 容器时，使用该网络地址为 Docker 容器分配 IP 地址。Docker 使用源码 network = addr.(*net.IPNet) 获取网桥设备的网络地址。

6.5.7 配置 Docker Daemon 的 iptables

创建完网桥之后，Docker Daemon 为容器以及宿主机配置 iptables，包括为容器之间所需要的 link 操作提供支持，为宿主机上所有的对外对内流量制定传输规则等。这部分详情可参见第 4 章，源码实现位于 ./docker/daemon/networkdriver/bridge/driver/driver.go#L133，如下所示：

```
// Configure iptables for link support
if enableIPTables {
    if err := setupIPTables(addr, icc); err != nil {
        return job.Error(err)
    }
}

// We can always try removing the iptables
if err := iptables.RemoveExistingChain("DOCKER"); err != nil {
    return job.Error(err)
}

if enableIPTables {
    chain, err := iptables.NewChain("DOCKER", bridgeIface)
    if err != nil {
        return job.Error(err)
    }
    portmapper.SetIptablesChain(chain)
}
```

6.5.8 配置网络设备间数据报转发功能

默认情况下，Linux 操作系统上的数据包转发功能是禁止的。数据包转发就是当宿主机存在多个网络设备时，如果其中一个接收到数据包，并将其转发给另外的网络设备。Docker Daemon 通过修改 /proc/sys/net/ipv4/ip_forward 的值，将其置为 1，则可以保证系统内数据包可以实现转发功能，代码如下：

```
if ipForward {
    // Enable IPv4 forwarding
    if err := ioutil.WriteFile("/proc/sys/net/ipv4/ip_forward", []byte{'1',
    '\n'}, 0644); err != nil {
        job.Logf("WARNING: unable to enable IPv4 forwarding: %s\n", err)
    }
}
```

6.5.9　注册网络 Handler

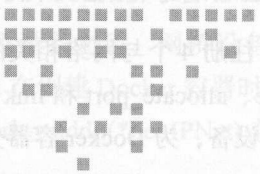
创建 Docker Daemon 网络环境的最后一个步骤是：注册 4 个与网络相关的 Handler。这 4 个处理方法分别是 allocate_interface、release_interface、allocate_port 和 link，作用分别是为 Docker 容器分配 IP 网络地址，释放 Docker 容器网络设备，为 Docker 容器分配端口资源，以及在 Docker 容器之间执行 link 操作。

至此，Docker Daemon 的网络环境初始化工作全部完成。

6.6　总结

在工业界，Docker 的网络问题备受关注。Docker 的容器技术以及镜像技术，已经给 Docker 实践者带来了诸多效益。然而 Docker 网络的发展依然具有很大的潜力，依然值得大家不断探索并推动其发展。

Docker 的网络环境可以分为 Docker Daemon 网络和 Docker Container 网络。本章从 Docker Daemon 的网络入手，分析了大家熟知的 Docker 桥接模式。



Chapter 7

第 7 章 Docker 容器网络

7.1 引言

如今，Docker 技术风靡全球，大家在尝试以及玩转 Docker 的同时，肯定离不开一个概念，那就是“容器”或者“ Container”。那么我们首先从原理的角度一窥“容器”或者“Container”的究竟。

第一次接触 Docker 时，大家肯定会深深感受到：在 Docker 容器内部运行应用程序竟是如此方便。第一，应用程序在 Docker 容器内部的部署与运行非常便捷，只要有 Dockerfile，应用一键式的部署绝对不是天方夜谭；第二，Docker 容器内运行的应用程序还可以受到资源的控制与隔离，大大满足云计算时代应用的要求。

毋庸置疑，Docker 的一些特性，的确摒弃了传统情况下开发模式的不少弊端。然而，这强大的功能背后到底是何等技术在“作祟”，又是谁可以支撑 Docker 的运行并提供丰富的特性？答案很简单，那就 Linux 内核。

那我们就从 Linux 内核的角度来看看 Docker 到底为何物，先从 Docker 容器入手。对于 Docker 容器，体验过的开发者都有两点非常重要的感受：内部可以跑应用（进程），以及提供隔离的环境。当然，后者肯定也是工业界称之为“容器”的原因之一。

首先，我们先来看 Docker 容器与进程的关系，或者容器与进程的关系。为了理清这两者之间的关系，我不妨提出这样一个问题“容器是否可以脱离进程而存在”。换句话说就是，能否创建一个容器，而这个容器内部没有任何进程。答案是否定的。既然答案是否定的，说明不可能先有容器，然后再有进程，那么问题又来了，“容器和进程是一起诞生的，还是先有进程再有容器呢？”可以说答案是后者。以下将慢慢阐述其中的原因。

阐述问题“容器是否可以脱离进程而存在”的原因前，相信大家对于下面这段话不会有异议：通过 Docker 创建出的一个 Docker Container 是一个容器，而这个容器提供了进程组隔离的运行环境。那么问题在于，容器到底是通过什么途径来实现进程组运行环境的“隔离”呢？此时，就轮到 Linux 内核隆重登场了。

说到运行环境的“隔离”，相信大家肯定对 Linux 内核中的 namespaces 和 cgroups 略有耳闻，namespaces 主要负责命名空间的隔离，cgroups 主要负责资源使用的限制。其实，正是这两个神奇的内核特性联合使用，才保证了 Docker 容器之间的“隔离”。那么，namespaces 和 cgroups 又和进程有什么关系呢？问题的答案可以用以下的次序来表示：

1）父进程通过 fork 创建子进程时，使用 namespaces 技术，实现子进程与父进程以及其他进程之间命名空间的隔离。

2）子进程创建完毕之后，使用 cgroups 技术来处理进程，实现进程的资源限制。

3）namespaces 和 cgroups 这两种技术都用上之后，进程所处的“隔离”环境才真正建立，此时“容器”真正诞生！

从 Linux 内核的角度分析容器的诞生，精简的流程即如上三步，而这三个步骤也恰好巧妙地阐述了 namespaces 和 cgroups 这两种技术和进程的关系，以及进程与容器的关系。进程与容器的关系，自然是：容器不能脱离进程而存在，先有进程，后有容器。然而，大家往往会说到“使用 Docker 创建 Docker 容器，然后在容器内部运行进程”。对此，从通俗易懂的角度来讲，这完全可以理解，因为“容器”一词的存在，本身就较为抽象。如果需要更为准确，可以表述为“使用 Docker 创建一个进程，为这个进程创建隔离的环境，这样的环境可以称为 Docker 容器，然后再在容器内部运行用户应用进程。”在这里，笔者的本意不是希望纠正外界对于 Docker 容器的认识，而是希望能和读者一起来看看 Docker 容器技术实现的原理到底几何。

更清楚地认识 Docker 容器之后，很快大家的眼球肯定会定位到 namespaces 和 cgroups 这两种技术上。Linux 内核的这两种技术，竟然起到如此重大的作用。那么下面我们就从 Docker 容器实现流程的角度简要介绍这两者。

首先讲述一下 namespace 在容器创建时的用法。用户启动容器，Docker Daemon 会 fork 出容器中的第一个进程 A（暂且称为进程 A，也就是 Docker Daemon 的子进程），执行 fork 时通过 5 个参数标志 CLONE_NEWNS、CLONE_NEWUTS、CLONE_NEWIPC、CLONE_PID 和 CLONE_NEWNET（Docker 1.2.0 还没有完全支持 user namespace）。Clone 系统调用一旦传入了这些参数标志，子进程将不再与父进程共享相同的命名空间（namespaces），而是由 Linux 为子进程创建新的命名空间（namespaces），从而保证子进程与父进程使用隔离的环境。另外，如果子进程 A 再次 fork 出子进程 B，而 fork 时没有传入相应的 namespaces 参数标志时，子进程 B 将会与 A 共享相同的命名空间（namespaces）。如果 Docker Daemon 再次创建一个 Docker 容器，内部进程有 D、E 和 F，那么这三个进程也会处于另外全新的 namespaces 中。两个容器的 namespaces 均与 Docker Daemon 所在的 namespaces 不同。Docker 中 Docker

Daemon 与 Docker 容器之间的 namespaces 关系，如图 7-1 所示。

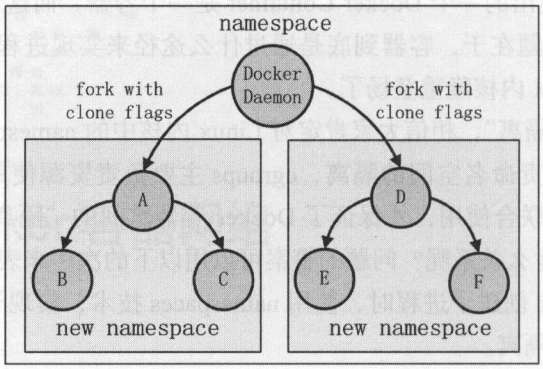


图 7-1　Docker 的 namespaces 关系图

再说起 cgroups，大家都知道可以使用 cgroups 为进程组做资源的限制。与 namespaces 不同的是，cgroup 的使用并不是在创建容器内进程时完成，而是在创建容器内进程之后完成，最终使得容器进程处于资源控制的状态。换言之，cgroups 的运用必须要等到容器内第一个进程被真正创建出来之后才能实现。当容器内进程创建完毕，Docker Daemon 可以获知容器内主进程的 PID 信息，随后将该 PID 放置在 cgroups 文件系统的指定位置，做相应的资源限制。如此一来，当容器主进程再 fork 新的子进程时，新的子进程同样受到与主进程相同的资源限制，效果就是整个进程组受到资源限制。

可以说 Linux 内核的 namespaces 和 cgroups 技术，实现了资源的隔离与限制。那么对于资源的隔离，是否还需要为容器准备必需的资源，比如说容器需要使用的网络资源，容器需要使用的文件系统挂载点等。这回答案是肯定的。网络资源就是一个很好的例子。当 Docker Daemon 为进程创建完隔离的运行环境之后，我们可以发现进程并没有独立的网络栈可以使用，如独立的网络接口等。此时，Docker Daemon 会将 Docker 容器内部所需要的资源一一配备齐全。网络方面，即为 Docker 容器通过用户指定的网络模式，配置相应的网络资源。

本章从源码的角度，分析 Docker 容器从无到有的过程中，Docker 容器网络创建的来龙去脉。简化而言，Docker 容器网络的创建流程如图 7-2 所示。

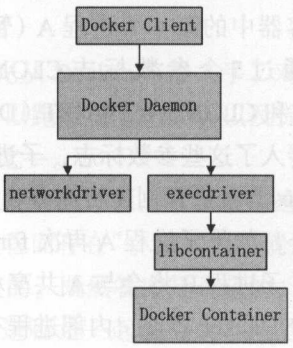


图 7-2　Docker 容器网络创建流程图

本章分析的主要内容有以下 5 部分：

- ❑ Docker 容器的网络模式
- ❑ Docker Client 配置容器网络
- ❑ Docker Daemon 创建容器网络流程
- ❑ execdriver 网络执行流程
- ❑ libcontainer 实现内核态网络配置

在 Docker 容器的网络创建过程中，networkdriver 模

块使用并非是重点，故分析内容中不涉及 networkdriver。不少读者肯定会有一些疑惑，为什么 Docker 容器的网络创建很少用到 networkdriver。这里强调一下 networkdriver 在 Docker 中的作用：第一，为 Docker Daemon 创建网络环境的时候，初始化 Docker Daemon 的网络环境（详见第 6 章），比如创建 docker0 网桥等；第二，为 Docker 容器分配 IP 地址，为 Docker 容器做端口映射等。而与 Docker 容器网络创建有关的内容并不多，如只在桥接模式下，为 Docker 容器的网络接口设备分配一个 IP 地址。

7.2　Docker 容器网络模式

如前所述，Docker 可以为容器创建隔离的网络环境，在隔离的网络环境下，Docker 容器使用独立的网络栈。看到这，很多读者也许会非常赞成以上的言论。然而，Docker 容器网络的高级特性又会让爱好者们感受到其中暗藏的更多玄机。

直奔主题，其实 Docker 除可以为 Docker 容器创建隔离的网络环境之外，同样有能力为 Docker 容器创建共享的网络环境。换言之，当开发者需要 Docker 容器与宿主机或者其他容器网络隔离，Docker 可以满足这样的需求；当开发者需要 Docker 容器的网络处于共享的网络环境时，Docker 同样可以满足这样的需求，并且网络共享，可以是 Docker 容器与宿主机之间网络共享，也可以是 Docker 容器与其他容器之间网络共享。神奇的是，Docker 还可以实现不为 Docker 容器创建网络环境。

通过以上描述，可以总结出 Docker 容器共有以下 4 种网络模式：bridge 桥接模式、host 模式、other container 模式和 none 模式。下面分别介绍 Docker 的 4 种网络模式。

7.2.1　bridge 桥接模式

Docker 容器的 bridge 桥接模式是目前 Docker 开发者中使用最为广泛的网络模式。bridge 桥接模式可以使 Docker 容器独立使用网络栈，或者说只有在容器内部的进程，才能使用该网络栈。bridge 桥接模式的实现步骤如下。

1）利用 veth pair 技术，在宿主机上创建两个虚拟网络接口，假设为 veth0 和 veth1。而 veth pair 技术的特性可以保证无论哪一个 veth 接收到网络报文，都会将报文传输给另一方。

2）Docker Daemon 将 veth0 附加到 Docker Daemon 创建的 docker0 网桥上。保证宿主机的网络报文有能力发往 veth0。

3）Docker Daemon 将 veth1 添加到 Docker 容器所属的网络命名空间（namespaces）下，veth1 在 Docker 容器看来就是 eth0。一方面，保证宿主机的网络报文若发往 veth0，可以立即被 veth1 收到，实现宿主机到 Docker 容器之间网络的联通性；另一方面，保证 Docker 容器单独使用 veth1，实现容器之间以及容器与宿主机之间网络环境的隔离性。

Docker 容器的 bridge 桥接模式如图 7-3 所示。

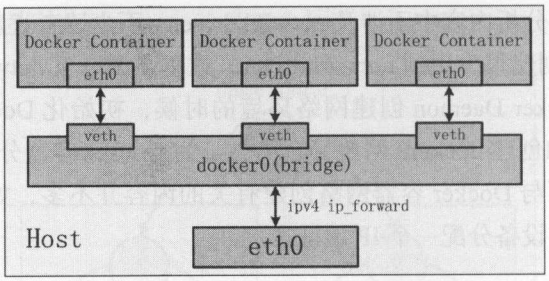


图 7-3 Docker 容器 bridge 桥接模式示意图

bridge 桥接模式，从原理上实现了 Docker 容器到宿主机乃至其他机器的网络联通性。然而，由于宿主机的 IP 地址与 veth pair 的 IP 地址不属于同一个网段，故仅仅依靠 veth pair 和网络命名空间的技术，还不足以使宿主机以外的网络主动发现 Docker 容器的存在。为使 Docker 容器有能力让宿主机以外的世界感受到容器内部暴露的服务，Docker 采用 NAT（Network Address Translation，网络地址转换）的方式让宿主机以外的世界可以将网络报文发送至容器内部。简要来讲，当 Docker 容器需要暴露服务时，内部服务必须监听容器 IP 和容器内部端口号 port_0，以便外界主动发起访问请求。由于宿主机以外的世界，只知道宿主机 eth0 的网络地址，并不知道 Docker 容器的 IP 地址，哪怕就算知道 Docker 容器的 IP 地址，从二层网络的角度来讲，外界也无法直接通过 Docker 容器的 IP 地址访问容器内部的服务。因此，Docker 使用 NAT 方法，将容器内部的服务与宿主机的某一个端口 port_1 进行“绑定”。

如此一来，外界访问 Docker 容器内部服务的流程为：

1）外界访问宿主机的 IP 以及宿主机的端口 port_1。

2）当宿主机接收到这类请求之后，由于存在 DNAT 规则，会将该请求的目的 IP（宿主机 eth0 的 IP）和目的端口 port_1 进行替换，替换为容器 IP 和容器端口 port_0。

3）由于能够识别容器 IP，故宿主机可以将请求发送给 veth pair。

4）veth pair 将请求发送至容器，容器交于内部服务进行处理。

使用 DNAT 方法，可以使 Docker 宿主机以外的世界主动访问 Docker 容器。那么 Docker 容器如何访问宿主机以外的世界呢？以下简要分析 Docker 容器内部访问宿主机以外世界的流程：

1）Docker 容器内部进程获悉宿主机外部服务的 IP 地址和端口 port_2，于是 Docker 容器发起请求。容器独立的网络环境保证了请求中报文的源 IP 地址为容器 IP（即容器内部 eth0，veth pair 一方的 IP 地址），另外 Linux 内核会自动为进程分配一个可用端口（假设为 port_3）。

2）请求通过容器内部 eth0 发送至 veth pair 的另一端，也就是到达网桥 docker0 处。

3）docker0 网桥开启了数据报转发功能（/proc/sys/net/ipv4/ip_forward），故 docker0 将请求发送至宿主机的 eth0 处。

4）宿主机处理请求时，使用 SNAT 对请求进行源地址 IP 替换，即将请求中源地址 IP(容

器 eth0 的 IP 地址）替换为宿主机 eth0 的 IP 地址。

5）宿主机将经过 SNAT 处理后的报文通过请求的目的 IP 地址（宿主机以外世界的 IP 地址）发送至外界。

在这里，很多人肯定要问：对于 Docker 容器内部主动发起对外的网络请求，请求到达宿主机进行 SNAT 处理发给外界之后，当外界响应请求时，响应报文中的目的 IP 地址肯定是 Docker Daemon 所在宿主机的 IP 地址，那响应报文回到宿主机的时候，宿主机又是如何转给 Docker 容器的呢？关于这样的响应，由于没有做相应的 DNAT 转换，原则上不会被发送至容器内部。为什么说对于这样的响应，不会做 DNAT 转换呢。原因很简单，DNAT 转换是针对特定容器内部服务监听的特定端口做的，该端口是供服务监听使用，而容器内部发起的请求报文中，源端口号肯定不会占用服务监听的端口，故容器内部发起请求的响应不会在宿主机上经过 DNAT 处理。

其实，这一环节的关键在于 iptables，具体的 iptables 规则如下：

```
iptables -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j
ACCEPT
```

此 iptables 规则的意思是：在宿主机上发往 docker0 网桥的网络数据报文，如果该数据报文所处的连接已经建立，则无条件接受，并由 Linux 内核将其转至原来的连接上，即回到 Docker 容器内部。

以上便是 Docker 容器中 bridge 桥接模式的简要介绍。可以说，bridger 桥接模式实现了两个方面：第一，让容器内部使用独立的网络栈；第二，让容器和宿主机以外的世界通过 NAT 建立通信。从功能的角度来讲，恰似具备了传统情况下隔离环境中网络的功能。然而，从使用的角度来讲，这种方式还没有弥合传统网络环境与容器网络环境的缝隙，换言之，NAT 方式必须使得容器服务在宿主机上有一层抽象，这层抽象需要 Docker 使用者人为的干涉，另外使用 NAT 方式，仅仅是三层网络上的实现手段，影响网络传输效率的同时，也会为网络的隔离带来不便。

7.2.2　host 模式

Docker 容器中的 host 模式与 bridge 桥接模式是完全不同的模式。最大的区别是：host 模式并没有为容器创建一个隔离的网络环境。之所以称之为 host 模式，是因为该模式下的 Docker 容器会和 host 宿主机使用同一个网络命名空间，故 Docker 容器可以和宿主机一样，使用宿主机的 eth0 和外界进行通信。如此一来，Docker 容器的 IP 地址自然也是宿主机 eth0 的 IP 地址。

Docker 容器的 host 网络模式如图 7-4 所示。

图 7-4 中最左侧的 Docker 容器，即采用了 host 网络模式，而其他两个 Docker 容器依然沿用 brdige 桥接模式，两种模式存在于同一个宿主机上并不矛盾。

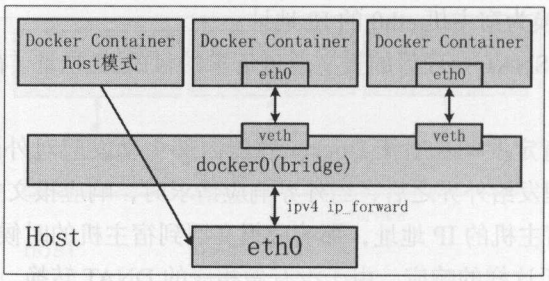


图 7-4 Docker 容器 host 网络模式示意图

Docker 容器的 host 网络模式在实现过程中，由于不需要额外的网桥以及虚拟网卡，故不会涉及 docker0 以及 veth pair。上文 namespace 的介绍中曾经提到：父进程在创建子进程的时候，如果不使用 CLONE_NEWNET 这个参数标志，那么创建出的子进程会与其父进程共享同一个网络命名空间。Docker 就是采用了这个简单的原理，在创建进程启动容器的过程中，没有传入 CLONE_NEWNET 参数标志，实现 Docker 容器与宿主机共享同一个网络环境，网络模式即为 host 网络模式。

Docker 容器的网络模式中，host 模式是 bridge 桥接模式很好的补充。采用 host 模式的 Docker 容器，可以直接使用宿主机的 IP 地址与外界进行通信，若宿主机的 eth0 是一个公有 IP，那么容器的 IP 也拥有这个公有 IP。同时容器内服务的端口也可以使用宿主机的端口，无需额外进行 NAT 转换。有这样的方便，自然会折损部分其他的特性，最明显的即是 Docker 容器网络环境的隔离性。另外，使用 host 模式的 Docker 容器虽然可以让容器内部的服务无差别、无改造的使用，但是由于网络的非隔离性，一旦宿主机上多个 Docker 容器都采用了 host 模式，并且内部的服务都需要占用相同的宿主机端口资源，此时会造成宿主机上端口资源的竞争，势必影响容器的服务质量。

7.2.3 other container 模式

Docker 容器的 other container 网络模式是 Docker 中一种较为特别的网络模式。之所以称为“ other container 模式”，是因为这个模式下的 Docker 容器，会使用其他容器的网络环境。之所以称为“特别”，是因为这个模式下容器的网络隔离性会处于 bridge 桥接模式与 host 模式之间。Docker 容器共享其他容器的网络环境，则至少这两个容器之间不存在网络隔离，而这两个容器又与宿主机以及除此之外其他的容器存在网络隔离。

Docker 容器的 other container 网络模式如图 7-5 所示。

图 7-5 中右侧的 Docker 容器即采用了 other container 网络模式，它具体能感受到网络环境即为左侧 Docker 容器的 bridge 桥接模式。

在 Docker 容器的 other container 网络模式实现过程中，不涉及网桥，同样也不需要创建虚拟网卡 veth pair。完成 other container 网络的创建只需要两个步骤：

1）查找 other container（即需要被共享网络环境的容器）的网络命名空间。

2）新创建的 Docker 容器的网络命名空间使用其他容器的网络命名空间。

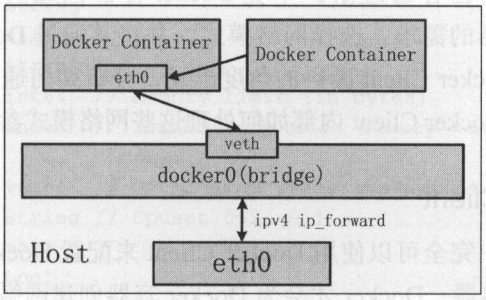


图 7-5　Docker 容器 other container 网络模式示意图

Docker 容器的 other container 网络模式，可以用来更好地服务于容器间的通信。需要互相通信的 Docker 容器在这种模式下，虽然网络隔离性没有 bridge 桥接模式优秀，但是相比 host 模式自然要好不少，同时网络隔离性稍有弱化的背后，带来的却是容器间高效的传输效率，以及容器间互访时简约的网络配置，即只需要使用 localhost 来访问其他容器。同时，多个容器共享同一个网络命令空间的形式，使得这部分容器的粘性大大提高。在容器的大规模调度与管理中，容器组必然会是一个需要攻克的技术点，而 other container 网络模式似乎为容器组的实现提供了方向。值得思考的是，Google 推出的 Kubernetes 平台中的 Pod 的原理，即使用了容器间共享网络命名空间的思路，实现了组的概念。

7.2.4　none 模式

Docker 容器的第 4 种网络模式是 none 模式。顾名思义，网络环境为 none，即不为 Docker 容器创建任何的网络环境。一旦 Docker 容器采用了 none 网络模式，那么容器内部就只能使用 loopback 网络接口，不会再有其他的网络资源。

可以说 none 模式为 Docker 容器做了极少的网络设定，但是俗话说得好“少即是多”，在没有网络配置的情况下，作为 Docker 开发者，才能在这种模式下做无限多可能的网络定制开发。这也恰恰体现了 Docker 开放的设计理念。

至此，Docker 的 4 种网络模式的介绍就告一段落，下文带来 Docker 网络模式的创建流程分析。

7.3　Docker Client 配置容器网络模式

Docker 容器网络模式的多样性，极大地满足了 Docker 用户部署应用的网络需求。Docker 用户，或者基于 Docker 的二次开发者，均可以在这基础上，选择使用与自身应用最

为贴切的网络模式。

从图 7-2 中可以看到，Docker 容器网络创建流程中第一个涉及的 Docker 模块就是 Docker Client。当然，这也容易理解，毕竟 Docker 容器网络环境的创建需要由用户发起。用户根据自身对容器的需求，选择网络模式，并将其通过 Docker Client 传递给 Docker Daemon。本小节将从 Docker Client 源码的角度出发，分析如何通过参数的形式配置 Docker 容器的网络模式，以及 Docker Client 内部如何处理这些网络模式参数。

7.3.1 使用 Docker Client

Docker 架构中，用户完全可以使用 Docker Client 来配置 Docker Container 的网络模式。实际情况下，启动一个容器，Docker 才会为 Docker 容器创建网络环境，故配置网络模式在用户执行 “启动容器” 命令之后完成。启动容器命令为 docker run，通过 Docker Client 发起，使用方式如下所示（其中 NETWORKMODE 为四种网络模式之一，IMAGE_NAME 为镜像名称）：

```
docker run -it--net NETWORKMODE IMAGE_NAME /bin/bash
```

执行以上命令时，Docker 首先创建一个 Docker Client，再由 Docker Client 解析整条命令的请求内容，最终解析为 run 命令，并发送至 Docker Daemon。Docker Client 的相关实现详见本书第 2 章。

7.3.2 runconfig 包解析

解析出 run 命令之后，Docker Client 调用相应的处理函数 CmdRun 处理关于 run 请求的具体内容。CmdRun 函数的实现位于 ./docker/docker/api/client/commands.go#L1990。CmdRun 执行的第一步就是：通过 runconfig 包中的 ParseSubcommand 函数执行解析出相应的 config、hostConfig，以及 cmd 对象，源码实现如下：

```
config, hostConfig, cmd, err := runconfig.ParseSubcommand(cli.Subcmd("run",
"[OPTIONS] IMAGE [COMMAND] [ARG...]", "Run a command in a new container"),
args, nil)
```

其中，config 的类型为 Config 结构体，hostConfig 的类型为 HostConfig 结构体，两种类型的定义均位于 runconfig 包。Config 与 HostConfig 类型同用以描述 Docker 容器的配置信息，然而两者之间又有着本质的区别，它们各自的描述如下：

- Config 结构体：描述 Docker 容器独立的配置信息。独立的含义是：Config 这部分信息描述的是容器本身，而不会与容器所在宿主机相关。
- HostConfig 结构体：描述 Docker 容器与宿主机相关的配置信息。

1. Config 结构体

Config 结构体描述 Docker 容器本身的属性信息，而结构体内的所有属性正是说明了这一

点，结构体的定义如下：

```
type Config struct {
    Hostname        string
    Domainname      string
    User            string
    Memory          int64  // Memory limit (in bytes)
    MemorySwap      int64  // Total memory usage (memory + swap); set '-1' to disable
                          //swap
    CpuShares       int64  // CPU shares (relative weight vs. other containers)
    Cpuset          string // Cpuset 0-2, 0,1
    AttachStdin     bool
    AttachStdout    bool
    AttachStderr    bool
    PortSpecs       []string  // Deprecated - Can be in the format of 8080/tcp
    ExposedPorts    map[nat.Port]struct{}
    Tty             bool   // Attach standard streams to a tty, including stdin if
                           // it is not closed.
    OpenStdin       bool   // Open stdin
    StdinOnce       bool   // If true, close stdin after the 1 attached client
                           // disconnects.
    Env             []string
    Cmd             []string
    Image           string // Name of the image as it was passed by the
                           // operator (eg. could be symbolic)
    Volumes         map[string]struct{}
    WorkingDir      string
    Entrypoint      []string
    NetworkDisabled bool
    OnBuild         []string
}
```

Config 结构体中各属性的详细说明如表 7-1 所示。

表 7-1　Config 结构体属性介绍表

Config 结构体属性名	类型	代表含义
Hostname	string	容器主机名
Domainname	string	域名服务器名称
User	string	容器内用户名
Memory	int64	容器的内存使用上限（单位：字节）
MemorySwap	int64	容器所有的内存使用上限（物理内存 + 交互区），关闭交互区支持置为 -1
CpuShares	int64	容器 CPU 使用 share 值，其他容器的相对值
Cpuset	string	CPU 核的使用集合
AttachStdin	bool	是否附加标准输入
AttachStdout	bool	是否附加标准输出
AttachStderr	bool	是否附加标准出错

（续）

Config 结构体属性名	类型	代表含义
PortsSpecs	[]string	已弃用
ExposedPorts	map[nat.Port]struct{}	容器内部暴露的端口号
Tty	bool	是否分配一个伪终端 tty
OpenStdin	bool	在没有附加标准输入时，是否依然打开标准输入
StdinOnce	bool	若为真，表示当一个客户关闭标准输入后关闭容器的标准输入
Env	[]string	容器的环境变量，可以有多个
Cmd	[]string	容器内运行的指令（一个或多个）
Image	string	容器 rootfs 所依赖的镜像名称
Volumes	map[string]struct{}	容器从宿主机上挂载的目录
WorkingDir	string	容器内部进程的指定工作目录
Entrypoint	[]string	覆盖镜像中默认的 ENTRYPOINT
NetworkDisabled	bool	是否关闭容器网络功能
OnBuild	[]string	指定的命令在构建镜像时不执行，而是在镜像构建完成之后被触发执行

2. HostConfig 结构体

HostConfig 结构体描述 Docker Container 与宿主机相关的属性信息，结构体的定义如下：

```
type HostConfig struct {
    Binds              []string
    ContainerIDFile    string
    LxcConf            []utils.KeyValuePair
    Privileged         bool
    PortBindings       nat.PortMap
    Links              []string
    PublishAllPorts    bool
    Dns                []string
    DnsSearch          []string
    VolumesFrom        []string
    Devices            []DeviceMapping
    NetworkMode        NetworkMode
    CapAdd             []string
    CapDrop            []string
    RestartPolicy      RestartPolicy
}
```

Config 结构体中各属性的详细说明如表 7-2 所示。

表 7-2 HostConfig 结构体属性介绍表

HostConfig 结构体属性名	类型	代表含义
Binds	[]string	从宿主机上绑定到容器的 volumes
ContainerIDFile	string	文件名，文件用以写入容器的 ID

（续）

HostConfig 结构体属性名	类型	代表含义
LxcConf	[]utils.KeyValuePair	添加自定义的 lxc 选项键值对
Privileged	bool	是否将容器设置为特权模式
PortBindings	nat.PortMap	容器绑定到宿主机的端口
Links	[]string	与其他容器之间的 link 信息
PublishAllPorts	bool	是否在宿主机上映射容器所有的端口信息
Dns	[]string	自定义的 DNS 服务器地址
DnsSearch	[]string	自定义的 DNS 查找服务器地址
VolumesFrom	[]string	将自身 volumes 挂载到此容器的容器
Devices	[]DeviceMapping	为容器添加一个或多个宿主机设备
NetworkMode	NetworkMode	为容器设置的网络模式
CapAdd	[]string	为容器用户添加一个或多个 Linux Capabilities
CapDrop	[]string	为容器用户禁用一个或多个 Linux Capabilities
RestartPolicy	RestartPolicy	当一个容器退出时采取的重启策略

3. runconfig 解析网络模式

分析完 Config 与 HostConfig 结构体之后，回到 runconfig 包中分析 Docker Client 如何解析与 Docker 容器网络模式相关的配置信息，并将这部分信息传递给 config 实例与 hostConfig 实例。

runconfig 包中的 ParseSubcommand 函数调用 parseRun 函数完成命令请求的分析，实现代码位于 ./docker/docker/runconfig/parse.go#L37-L39，如下所示：

```
func ParseSubcommand(cmd *flag.FlagSet, args []string, sysInfo *sysinfo.SysInfo)
(*Config, *HostConfig, *flag.FlagSet, error) {
    return parseRun(cmd, args, sysInfo)
}
```

进入 parseRun 函数即可发现，该函数完成了四方面的工作：

- 定义与容器配置信息相关的 flag 参数。
- 解析 docker run 命令后紧跟的请求内容，将请求内容全部保存至 flag 参数中。
- 通过 flag 参数验证参数的有效性，并处理得到 Config 结构体与 HostConfig 结构体需要的属性值。
- 创建并初始化 Config 类型实例 config、HostConfig 类型实例 hostConfig，最终返回 config、hostConfig 与 cmd。

本章分析 Docker 容器的网络模式，而 parseRun 函数中有关容器网络模式的 flag 参数有 flNetwork 与 flNetMode，两者的定义分别位于 ./docker/docker/runconfig/parse.go#L62 与 ./docker/runconfig/parse.go#L75，如下所示：

```
flNetwork = cmd.Bool([]string{"#n", "#-networking"}, true, "Enable networking
```

```
for this container")
flNetMode = cmd.String([]string{"-net"}, "bridge", "Set the Network mode for
the container\n'bridge': creates a new network stack for the container on the
docker bridge\n'none': no networking for this container\n'container:<name|id>':
reuses another container network stack\n'host': use the host network stack
inside the container.  Note: the host mode gives the container full access to
local system services such as D-bus and is therefore considered insecure.")
```

可见 flag 参数 flNetwork 表示是否开启容器的网络模式，若为 true 则开启，说明需要给容器创建网络环境；否则不开启，说明不给容器赋予网络功能。此 flag 参数的默认值为 true，另外使用该 flag 的方式为：在 docker run 之后设定 --networking 或者 -n，由于 flNetwork 的名称为 {"#n", "#-networking"}，故可以判定此 flag 已经处于弃用状态。

另一个 flag 参数 flNetMode 则表示用户为容器设定的网络模式，共有四种选项分别是：bridge、none、container:<name|id> 和 host。四种模式的作用上文已经有所介绍，此处不再赘述。使用此 flag 的方式为：在 docker run 之后设定 --net，如：

```
docker run -it --net host IMAGE_NAME /bin/bash
```

用户使用 docker run 启动容器时设定了以上两个 flag 参数（实际只有 flNetMode 一个 flag 参数），则 runconfig 包会解析出这两个 flag 的值。最终，通过 flag 参数 flNetwork，得到 Config 类型实例 config 的属性 NetworkDisabled；通过 flag 参数 flNetMode，得到 HostConfig 类型实例 hostConfig 的属性 NetworkMode。

函数 parseRun 返回 config、hostConfig 与 cmd，代表着 runconfig 包解析配置参数工作的完成，下一节将回到 CmdRun 的运行。

7.3.3 CmdRun 执行

在 runconfig 包中，Docker Client 已经将有关容器网络模式的配置置于 config 对象与 hostConfig 对象，故在 CmdRun 函数的执行中，更多的是基于 config 对象与 hostConfig 参数的配置信息处理，而没有其他的容器网络处理部分。

CmdRun 的主要工作是：利用 Docker Daemon 暴露的 RESTful API 接口，将 docker run 的请求发送至 Docker Daemon。CmdRun 执行过程中 Docker Client 与 Docker Daemon 的简易交互如图 7-6 所示。

从 CmdRun 的执行流程中，我们可以发现：在解析 config、hostConfig 与 cmd 之后，Docker Client 首先发起请求 create container。若 Docker Daemon 关于该容器的镜像存在，则立即执行 create container 操作并返回请求响应；Docker Client 收到响应后，再发起请求 start container。若容器镜像还不存在，Docker Daemon 返回一个 404 的错误，表示镜像不存在；Docker Client 收到错误响应之后，再发起一个请求 pull image，使 Docker Daemon 首先下载镜像，下载完之后 Docker Client 再次发起请求 create container，Docker Daemon 创建完之后，Docker Client 最终发起请求 start container。

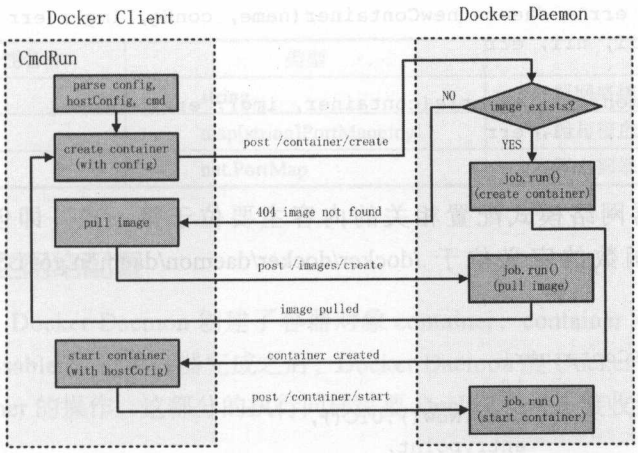


图 7-6　Docker Client 与 Docker Daemon 交互图

其中关于 Docker 容器网络模式的参数配置均存储于 config 与 hostConfig 对象之中，在请求 create container 和 start container 发起后，随请求一起发送至 Docker Daemon。

7.4　Docker Daemon 创建容器网络流程

Docker Daemon 接收到 Docker Client 的请求可以分为两次，第一次为 create container，第二次为 start container。这两次请求的执行过程，都与 Docker 容器的网络相关。以下按照这两个请求的执行，具体分析 Docker 容器网络的创建。Docker Daemon 如何通过 Docker Server 解析 RESTful 请求，并完成路由，在第 5 章已经详细分析过，故本章不再赘述。

7.4.1　创建容器之网络配置

Docker Daemon 创建容器主要执行了 create container 操作。

创建容器过程中，Docker Daemon 首先通过 runconfig 包中的 ContainerConfigFromJob 函数，解析出请求中的 config 对象，解析过程代码如下：

```
config := runconfig.ContainerConfigFromJob(job)
```

至此，Docker Client 处理得到的 config 对象，已经传递至 Docker Daemon 的 config 对象，config 对象中已经含有属性 NetworkDisabled 的具体值。

而容器创建的工作内容主要有以下两点：

1）创建与 Docker 容器对应的 Container 类型实例 container。

2）创建 Docker 容器的 rootfs。

具体的源码实现位于 ./docker/docker/daemon/create.go#L73-L78，如下所示：

```
if container, err = daemon.newContainer(name, config, img); err != nil {
    return nil, nil, err
}
if err := daemon.createRootfs(container, img); err != nil {
    return nil, nil, err
}
```

与 Docker 容器网络模式配置相关的内容主要位于第一点，即创建 container 实例中。newContainer 函数的定义位于 ./docker/docker/daemon/daemon.go#L516-L550，具体的 container 实例如下：

```
container := &Container{
    ID:              id,
    Created:         time.Now().UTC(),
    Path:            entrypoint,
    Args:            args, //FIXME: de-duplicate from config
    Config:          config,
    hostConfig:      &runconfig.HostConfig{},
    Image:           img.ID, // Always use the resolved image id
    NetworkSettings: &NetworkSettings{},
    Name:            name,
    Driver:          daemon.driver.String(),
    ExecDriver:      daemon.execDriver.Name(),
    State:           NewState(),
}
```

在 container 对象中，config 对象直接赋值给 container 对象的 Config 属性，另外 hostConfig 属性与 NetworkSettings 属性均为空。其中 hostConfig 对象将在 start container 请求执行过程中被赋值，NetworkSettings 类型的作用是描述容器的网络具体信息，定义位于 ./docker/docker/daemon/network_settings.go#L11-L18，源码如下所示：

```
type NetworkSettings struct {
    IPAddress   string
    IPPrefixLen int
    Gateway     string
    Bridge      string
    PortMapping map[string]PortMapping // Deprecated
    Ports       nat.PortMap
}
```

Networksettings 类型的各属性详细说明如表 7-3 所示。

表 7-3 NetworkSettings 属性介绍表

NetworkSettings 属性名称	类型	含义
IPAddress	string	容器网络接口的 IP 网络地址
IPPrefixLen	int	网络标识位长度
Gateway	string	容器的默认网关地址

（续）

NetworkSettings 属性名称	类型	含义
Bridge	string	容器网络接口使用的网桥地址
PortMapping	map[string]PortMapping	容器与宿主机的端口映射
Ports	nat.PortMap	容器内部暴露的端口号

7.4.2　启动容器之网络配置

创建容器阶段，Docker Daemon 创建了容器对象 container，container 对象内部的 Config 属性含有 NetworkDisabled。创建容器完成之后，Docker Daemon 应 Docker Client 的请求，需要执行 create container 的操作。这部分的执行同样需要 Docker Server 接收请求，并分发调度处理。

Docker Daemon 启动容器主要执行了 start container 操作。

启动容器过程中，Docker Daemon 首先通过 runconfig 包中的 ContainerHostConfigFromJob 函数，解析出请求中的 hostConfig 对象，解析过程源码如下：

```
hostConfig := runconfig.ContainerHostConfigFromJob(job)
```

至此，Docker Client 处理得到的 hostConfig 对象，已经传递至 Docker Daemon 的 hostConfig 对象，hostConfig 对象中已经含有属性 NetworkMode 具体值。

容器启动的所有工作，均由以下 Start 函数来完成，源码位于 ./docker/docker/daemon/start.go#L36-L38，如下所示：

```
if err := container.Start(); err != nil {
    return job.Errorf("Cannot start container %s: %s", name, err)
}
```

Start 函数实现了容器的启动。更为具体的描述是：Start 函数实现了进程的启动，另外在启动进程的同时为进程设定了命名空间（namespaces）并为进程做了资源的限制，从而保证进程以及之后进程的子进程都会在相同的命名空间内，且受到相同的资源控制。如此一来，Start 函数创建的进程，以及该进程之后的子进程，形成一个进程组，该进程组处于资源隔离和资源控制的环境中，我们习惯将这样的进程组环境称为容器，也就是这里的 Docker 容器。

回到 Start 函数的执行，位于 ./docker/docker/daemon/container.go#L275-L320。Start 函数的执行过程与 Docker 容器网络模式相关的主要有三部分：

- initializeNetwork，初始化 container 对象中与网络相关的属性。
- populateCommand，填充 Docker 容器内部需要执行的命令，Command 中含有进程启动命令，还含有容器环境的配置信息，也包括网络配置。
- container.waitForStart()，实现 Docker 容器内部进程的启动，进程启动之后，为容器创建网络环境等。

1. 初始化容器网络配置

容器对象 container 中有属性 hostConfig，属性 hostConfig 中有属性 NetworkMode，初始化容器网络配置 initializeNetworking() 的主要工作就是：通过 NetworkMode 属性为 Docker 容器的网络做相应的初始化配置工作。

Docker Container 的网络模式有四种，分别为：host、other container、none 以及 bridge。initializeNetworking 函数的执行完全覆盖了这四种模式。

initializeNetworking() 函数的源码实现位于 ./docker/docker/daemon/container.go#L881-L933，如下所示：

```
func (container *Container) initializeNetworking() error {
var err error
if container.hostConfig.NetworkMode.IsHost() {
     container.Config.Hostname, err = os.Hostname()
     if err != nil {
          return err
     }

     parts := strings.SplitN(container.Config.Hostname, ".", 2)
     if len(parts) > 1 {
          container.Config.Hostname = parts[0]
          container.Config.Domainname = parts[1]
     }

     content, err := ioutil.ReadFile("/etc/hosts")
     if os.IsNotExist(err) {
          return container.buildHostnameAndHostsFiles("")
     } else if err != nil {
          return err
     }

     if err := container.buildHostnameFile(); err != nil {
          return err
     }

     hostsPath, err := container.getRootResourcePath("hosts")
     if err != nil {
          return err
     }
     container.HostsPath = hostsPath

     return ioutil.WriteFile(container.HostsPath, content, 0644)
} else if container.hostConfig.NetworkMode.IsContainer() {
     // we need to get the hosts files from the container to join
     nc, err := container.getNetworkedContainer()
     if err != nil {
          return err
     }
```

```
        container.HostsPath = nc.HostsPath
        container.ResolvConfPath = nc.ResolvConfPath
        container.Config.Hostname = nc.Config.Hostname
        container.Config.Domainname = nc.Config.Domainname
    } else if container.daemon.config.DisableNetwork {
        container.Config.NetworkDisabled = true
        return container.buildHostnameAndHostsFiles("127.0.1.1")
    } else {
        if err := container.allocateNetwork(); err != nil {
            return err
        }
        return container.buildHostnameAndHostsFiles(container.NetworkSettings.IPAddress)
    }
    return nil
    }
```

针对以上源码，下面以 4 种不同的网络模式分析 initializeNetworking 函数的作用。

第一，host 网络模式。Docker 容器网络的 host 模式意味着容器使用宿主机的网络环境。虽然 Docker 容器使用宿主机的网络环境，但这并不代表 Docker 容器可以拥有宿主机文件系统的视角，而 host 宿主机上有很多信息标识的是网络信息，故 Docker Daemon 需要将这部分标识网络的信息，从宿主机上添加到 Docker 容器内部的指定位置。这样的网络信息，主要有以下三种：

- 宿主机的 hostname 文件，代表容器的主机名。
- 宿主机的 hosts 文件，代表容器的主机名配置文件。
- 宿主机 resolv.conf 文件，属于容器的域名文件。

由于 Docker 容器与宿主机共享网络，因此 Docker 容器的 hostname 文件、hosts 文件以及 resolv.conf 文件原则上与宿主机上的这些文件内容应该保持一致。结果就是：Docker Daemon 将宿主机的 hostname 文件、hosts 文件以及 resolv.conf 内容写入 Docker 容器的指定目录下。目录一般为 /var/lib/docker/containers/<container_id>，当容器启动时，再将这部分文件挂载进容器内部的指定路径。

第二，other container 网络模式。Docker 容器的 other container 网络模式意味着：容器使用其他已经创建容器的网络环境。

Docker Daemon 首先判断 host 网络模式，若不为 host 网络模式，则继续判断是否为 other container 模式。若 Docker 容器的网络模式为 other container（假设使用的 -net 参数为 --net=container:17adef，其中 17adef 为容器 ID），则用户必须指定被共享网络的 Docker 容器，此处容器 ID 为 17adef。在这种情况下，Docker Daemon 所做的执行操作包括两步。

第一步，从 container 对象的 hostConfig 属性中找出 NetworkMode，并找到相应的容器，即 17adef 的容器对象 container，实现源码如下：

```
nc, err := container.getNetworkedContainer()
```

第二步，将 17adef 容器对象的 HostsPath、ResolveConfPath、Hostname 和 Domainname 赋值给当前容器对象 container，实现源码如下：

```
container.HostsPath = nc.HostsPath
container.ResolvConfPath = nc.ResolvConfPath
container.Config.Hostname = nc.Config.Hostname
container.Config.Domainname = nc.Config.Domainname
```

第三，none 网络模式。Docker 容器的 none 网络模式意味着不给该容器创建任何网络环境，容器只能使用 127.0.1.1 的环回接口。

Docker Daemon 通过 config 属性的 DisableNetwork 来判断是否为 none 网络模式。实现源码如下：

```
if container.daemon.config.DisableNetwork {
    container.Config.NetworkDisabled = true
    return container.buildHostnameAndHostsFiles("127.0.1.1")
}
```

第四，bridge 网络模式。Docker 容器的 bridge 网络模式意味着为容器创建桥接网络模式。桥接模式使得 Docker 容器创建独立的网络环境，并通过“桥接”的方式实现 Docker 容器与外界的网络通信。

初始化 bridge 网络模式的配置，实现源码如下：

```
if err := container.allocateNetwork(); err != nil {
     return err

}
return container.buildHostnameAndHostsFiles(container.NetworkSettings.IPAddress)
```

以上代码完成的内容主要也是两部分：第一，通过 allocateNetwork 函数为容器分配一个网络接口可用的 IP 地址，并将网络配置（包括 IP、bridge、Gateway 等）赋值给 container 对象的 NetworkSettings；第二，通过 NetworkSettings 为容器创建 hosts、hostname 等文件。

2. 创建容器 Command 信息

Docker 在实现容器时，在源码层次使用了 Command 类型。Command 是一个非常重要的概念，我们可以认为 Command 类型包含了两部分的内容：第一，运行容器内进程的外部命令 exec.Cmd；第二，运行容器时启动进程需要的所有基础信息：包括容器进程组的使用资源、网络资源、使用设备、工作路径等。通过这两部分的内容，我们清楚如何创建容器内的进程，同时也清楚为容器创建什么样的环境。

首先，我们先来看 Command 类型的定义，位于 ./docker/docker/daemon/execdriver/driver.go#L84。通过分析 Command 类型以及相关的其他数据结构类型，得到的 Command 类型关系如图 7-7 所示。

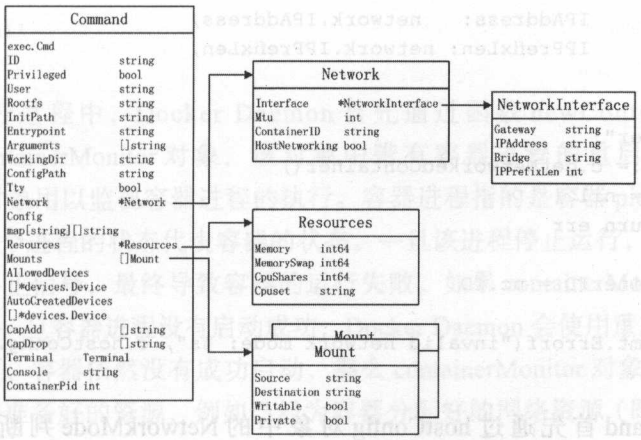


图 7-7　Command 类型关系图

从 Command 类型关系图中，我们可以看到 Command 类型中第一个属性为 exec.Cmd，即代表需要创建的进程具体的外部命令；同时，关于网络方面的属性有 Network，Network 的类型为 *Network；关于 Docker 容器资源使用方面的属性为 Resources，从 Resource 的类型来看，Docker 目前能做的资源限制有 4 个维度，分别为内存，内存 +Swap，CPU 使用，CPU 核使用；关于挂载的内容，有属性 Mounts；等等。

简单介绍 Command 类型之后，回到 Docker Daemon 启动容器网络的源码分析。紧接在函数 initializeNetworking 之后，是 populateCommand 环节。populateCommand 的函数实现位于 ./docker/docker/daemon/container.go#L191-L274。上文已经提及，populateCommand 的作用是创建 execdriver 包中的对象 Command 实例，该 Command 中既有启动容器进程的外部命令，同时也有众多容器环境的配置信息，包括网络。

本小节，分析 populateCommand 如何填充 Command 对象中的网络信息，其他内容的分析会在第 12 章和第 13 章进行展开。

Docker 容器有四种网络模式，故 populateCommand 首先需要判断容器属于哪种网络模式，随后将具体的网络模式信息，写入 Command 对象的 Network 属性中。查验 Docker 容器网络模式的源码位于 ./docker/docker/daemon/container.go#L204-L227，如下所示：

```
parts := strings.SplitN(string(c.hostConfig.NetworkMode), ":", 2)
switch parts[0] {
case "none":
case "host":
    en.HostNetworking = true
case "bridge", "": // empty string to support existing containers
    if !c.Config.NetworkDisabled {
        network := c.NetworkSettings
        en.Interface = &execdriver.NetworkInterface{
            Gateway:     network.Gateway,
            Bridge:      network.Bridge,
```

```
                IPAddress:   network.IPAddress,
                IPPrefixLen: network.IPPrefixLen,
            }
        }
    case "container":
        nc, err := c.getNetworkedContainer()
        if err != nil {
            return err
        }
        en.ContainerID = nc.ID
    default:
        return fmt.Errorf("invalid network mode: %s", c.hostConfig.NetworkMode)
    }
```

populateCommand 首先通过 hostConfig 对象中的 NetworkMode 判断容器属于哪种网络模式。该部分内容涉及 execdriver 包中的 Network，可参见 Command 类型关系图中的 Network 类型。若为 none 模式，则对于 Network 对象（即 en，*execdriver.Network）不做任何操作。若为 host 模式，则将 Network 对象的 HostNetworking 置为 true；若为 bridge 桥接模式，则首先创建一个 NetworkInterface 对象，完善该对象的 Gateway、Bridge、IPAddress 和 IPPrefixLen 信息，最后将 NetworkInterface 对象作为 Network 对象的中 Interface 属性的值；若为 other container 模式，则首先通过 getNetworkedContainer() 函数获知被分享网络命名空间的容器，最后将容器 ID，赋值给 Network 对象的 ContainerID。由于 bridge 模式、host 模式以及 other container 模式彼此互斥，故 Network 对象中 Interface 属性、ContainerID 属性以及 HostNetworking 三者之中只有一个被赋值。当 Docker 容器的网络被查验之后，populateCommand 将 en 实例 Network 属性的值，传递给 Command 对象。

3. 启动容器内部进程

当为容器做好所有的配置之后，Docker Daemon 需要真正意义上的启动容器。根据启动容器流程中涉及的 Docker 模块，Docker Daemon 后续环节中实现启动容器的请求会被发送至 execdriver，再经过 libcontainer，最后实现 Linux 内核级别的进程启动。

回到 Docker Daemon 的启动容器，daemon 包中 start 函数的最后一步即为执行 container.waitForStart()。waitForStart 函数的定义位于 ./docker/daemon/container.go#L1070-L1082，源码如下：

```
func (container *Container) waitForStart() error {
    container.monitor = newContainerMonitor(container, container.hostConfig.RestartPolicy)

    select {
    case <-container.monitor.startSignal:
    case err := <-utils.Go(container.monitor.Start):
        return err
    }
```

```
    return nil
}
```

以上源码运行过程中，Docker Daemon 首先通过函数 newContainerMonitor 返回一个初始化的 containerMonitor 对象，该对象中带有容器进程的重启策略。总体而言，containerMonitor 对象用以监视容器进程的执行。容器进程指的是容器 pid namespace 内进程号为 1 的进程，这个进程的状态代表容器的状态。一旦该进程停止运行，则容器内部所有进程都将收到一个终止信号，最终导致容器的运行失败。如果 containerMonitor 中指定了进程的重启策略，那么一旦容器进程没有启动成功，Docker Daemon 会使用重启策略来重启容器。如果在重启策略下，容器依然没有成功启动，那么 containerMonitor 对象会负责重置以及清除所有已经为容器准备好的资源，例如已经为容器分配好的网络资源（即 IP 地址），还有为容器准备的 rootfs 等。

waitForStart() 函数通过 container.monitor.Start 来实现容器的启动，进入 ./docker/docker/daemon/monitor.go#L100，可以发现启动容器进程位于 ./docker/docker/daemon/monitor.go#L136，源码如下：

```
exitStatus, err = m.container.daemon.Run(m.container, pipes, m.callback)
```

以上源码实际调用了 daemon 包中的 Run 函数，位于 ./docker/daemon/daemon.go#L969-L971，如下所示：

```
func (daemon *Daemon) Run(c *Container, pipes *execdriver.Pipes, startCallback
execdriver.StartCallback) (int, error) {
    return daemon.execDriver.Run(c.command, pipes, startCallback)
}
```

最终，Run 函数中调用了 execdriver 中的 Run 函数来执行 Docker Container 的启动命令。

至此，网络部分在 Docker Daemon 内部的执行已经结束，紧接着程序的运行陷入 execdriver，进一步运行容器启动的相关步骤。

7.5　execdriver 网络执行流程

Docker 架构中 execdriver 的作用是启动容器内部进程，最终启动容器。目前，在 Docker 中 execdriver 作为执行驱动，可以有两种选项：lxc 与 native。其中，lxc 驱动会调用 lxc 工具实现容器的启动，而 native 驱动会使用 Docker 官方发布的 libcontainer 来启动容器。

Docker Daemon 启动过程中，execdriver 的类型默认为 native，故本章主要分析 native 驱动在执行启动容器时，如何处理网络部分。

在 Docker Daemon 启动容器的最后一步，即调用了 execdriver 的 Run 函数来执行。通过分析 Run 函数的具体实现，可知关于 Docker 容器的网络执行流程主要包括两个环节：

1）创建 libcontainer 的 Config 对象。

2）通过 libcontainer 中的 namespaces 包执行启动容器。

将 execdriver.Run 函数的运行流程展开，与 Docker 容器网络相关的流程，如图 7-8 所示。

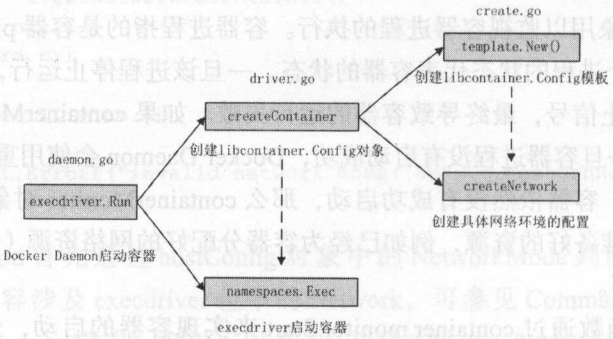


图 7-8 execdriver.Run 执行流程图

7.5.1 创建 libcontainer 的 Config 对象

Run 函数位于 ./docker/docker/daemon/execdriver/native/driver.go#L62-L168。进入 Run 函数的实现，立即可以发现该函数通过 createContainer 创建了一个 container 对象，源码如下：

```
container, err := d.createContainer(c)
```

其中 c 为 Docker Daemon 创建的 execdriver.Command 类型实例。以上源码中 createContainer 函数的作用是：使用 execdriver.Command 来填充 libcontainer.Config。简要介绍 libcontainer.Config 的作用就是，它定义了在一个容器化的环境中执行一个进程所需要的所有配置项。createContainer 函数使用 Docker Daemon 层创建的 execdriver.Command，创建更底层 libcontainer 所需要的 Config 对象。从这个角度来看，execdriver 更像是封装了 libcontainer 对外的接口，实现了将 Docker Daemon 特有的容器启动信息转换为底层 libcontainer 能真正使用的容器配置选项。libcontainer.Config 类型与其内部对象之间的关系如图 7-9 所示。

createContainer 的源码实现部分位于 ./docker/docker/daemon/execdriver/native/create.go#L23-L77，如下所示：

```
func (d *driver) createContainer(c *execdriver.Command) (*libcontainer.Config, error) {
    container := template.New()
    ...
    if err := d.createNetwork(container, c); err != nil {
        return nil, err
    }
    ...
    return container, nil
}
```

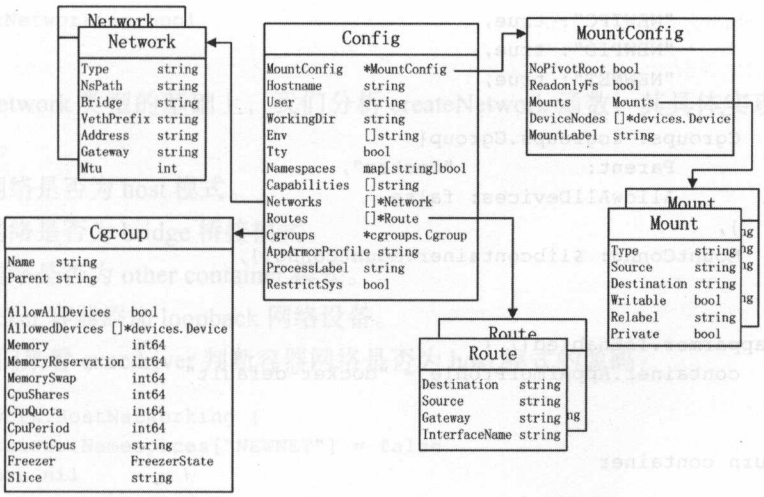


图 7-9　libcontainer.Config 类型关系图

1. libcontainer.Config 模板实例

从 createContainer 函数的实现以及 execdriver.Run 执行流程图中都可以看到，createContainer 所做的第一个操作就是执行 template.New()，意为创建一个 libcontainer.Config 的实例容器。其中，template.New() 的定义位于 ./docker/docker/daemon/execdriver/native/template/default_template.go，主要的作用为返回 libcontainer 关于 Docker 容器的默认配置选项。

Template.New() 的源码实现如下：

```
func New() *libcontainer.Config {
    container := &libcontainer.Config{
        Capabilities: []string{
            "CHOWN",
            "DAC_OVERRIDE",
            "FSETID",
            "FOWNER",
            "MKNOD",
            "NET_RAW",
            "SETGID",
            "SETUID",
            "SETFCAP",
            "SETPCAP",
            "NET_BIND_SERVICE",
            "SYS_CHROOT",
            "KILL",
            "AUDIT_WRITE",
        },
        Namespaces: map[string]bool{
            "NEWNS":  true,
            "NEWUTS": true,
```

```
            "NEWIPC": true,
            "NEWPID": true,
            "NEWNET": true,
        },
        Cgroups: &cgroups.Cgroup{
            Parent:          "docker",
            AllowAllDevices: false,
        },
        MountConfig: &libcontainer.MountConfig{},
    }

    if apparmor.IsEnabled() {
        container.AppArmorProfile = "docker-default"
    }

    return container
}
```

libcontainer.Config 默认的模板对象，首先设定了 Capabilities 的默认项，如 CHOWN、DAC_OVERRIDE、FSETID 等；其次又为 Docker 容器所需设定的 namespaces 添加默认值，即需要创建 5 个命名空间，如 NEWNS、NEWUTS、NEWIPC、NEWPID 和 NEWNET，其中不包括 user namespace，另外与网络相关的 namespace 是 NEWNET；最后设定了一些关于 cgroup 以及 apparmor 的默认配置。

Template.New() 函数最后返回类型为 libcontainer.Config 的实例 container，该实例中只包含默认配置项，其他内容的添加需要 createContainer 的后续代码来完成。

2. createNetwork 实现

在 createContainer 的实现流程中，为了完善 container 对象（类型为 libcontainer.Config），后续还有很多步骤，如与网络相关的 createNetwork 函数调用，与 Linux 内核 Capabilities 相关的 setCapabilities 函数调用，与 cgroups 相关的 setupCgroups 函数调用，以及与挂载目录相关的 setupMounts 函数调用等。本小节主要分析 createNetwork 函数如何为 container 对象完善网络配置项。

createNetwork 函数的定义位于 ./docker/docker/daemon/execdriver/native/create.go#L79-L124，该函数主要利用 execdriver.Command 中 Network 属性的内容，来判断如何创建 libcontainer.Config 中的 Network 属性（关于两种 Network 属性，可以参见图 7-7 和图 7-9）。由于 Docker 容器的 4 种网络模式彼此互斥，故以上 Network 类型中 Interface、ContainerID 与 HostNetworking 最多只有一项会被赋值。

execdriver.Command 中 Network 的类型定义如下：

```
type Network struct {
    Interface       *NetworkInterface
    Mtu             int
    ContainerID     string
```

```
    HostNetworking bool
}
```

在以上 Network 类型的基础上，我们分析 createNetwork 函数，其具体实现可以归纳为以下 4 个部分：

1）判断网络是否为 host 模式。

2）判断网络是否为 bridge 桥接模式。

3）判断网络是否为 other container 模式。

4）为 Docker 容器添加 loopback 网络设备。

首先，我们来看 execdriver 判断容器网络是否为 host 模式的源码：

```
if c.Network.HostNetworking {
    container.Namespaces["NEWNET"] = false
    return nil
}
```

当 execdriver.Command 类型实例中 Network 属性的 HostNetworking 为 true，则说明需要为 Docker 容器创建 host 网络模式，使容器与宿主机共享相同的网络命名空间。在 host 模式的具体介绍中，我们已经阐明，只要 Docker Daemon 在创建容器进程，进行 CLONE 系统调用时，不传入 CLONE_NEWNET 参数标志即可实现共享网络命名空间。这部分源码正好准确地验证了这一点。Docker Daemon 将 container 对象中代表网络命名空间的 NEWNET 设为 false，最终导致 libcontainer 中不创建新的网络命名空间。

再来看，execdriver 判断容器网络是否为 bridge 桥接模式的源码：

```
if c.Network.Interface != nil {
    vethNetwork := libcontainer.Network{
        Mtu:        c.Network.Mtu,
        Address:      fmt.Sprintf("%s/%d", c.Network.Interface.IPAddress,
        c.Network.Interface.IPPrefixLen),
        Gateway:    c.Network.Interface.Gateway,
        Type:       "veth",
        Bridge:     c.Network.Interface.Bridge,
        VethPrefix: "veth",
    }
    container.Networks = append(container.Networks, &vethNetwork)
}
```

当 execdriver.Command 类型实例中 Network 属性的 Interface 不为 nil 值，则说明需要为 Docker 容器创建 bridge 桥接模式，使容器拥有隔离的网络环境。于是，以上源码为 libcontainer.Config 的 container 对象添加 Networks 属性 vethNetwork，网络接口类型为 veth，以便 libcontainer 在执行时，可以为 Docker 容器创建 veth pair。

接着来看 execdriver 判断容器网络是否为 other container 模式的代码：

```
if c.Network.ContainerID != "" {
```

```
        d.Lock()
        active := d.activeContainers[c.Network.ContainerID]
        d.Unlock()

        if active == nil || active.cmd.Process == nil {
             return fmt.Errorf("%s is not a valid running container to join",
             c.Network.ContainerID)
        }
        cmd := active.cmd

        nspath := filepath.Join("/proc", fmt.Sprint(cmd.Process.Pid), "ns", "net")
        container.Networks = append(container.Networks, &libcontainer.Network{
             Type:   "netns",
             NsPath: nspath,
        })
    }
```

当 execdriver.Command 类型实例中 Network 属性的 ContainerID 不为空字符串时，则说明需要为 Docker 容器创建 other container 模式，使创建容器共享其他容器的网络环境。实现过程中，execdriver 首先需要在 activeContainers 中查找需要被共享网络环境的容器 active；并通过 active 容器的启动执行命令 cmd 找到容器主进程在宿主机上的 PID；随后在 proc 文件系统中找到该进程 PID 的关于网络命名空间的路径 nspath，也是整个容器的网络命名空间路径；最后为类型为 libcontainer.Config 的 container 对象添加 Networks 属性，Network 的类型为 netns。

此外，createNetwork 函数还实现了为 Docker 容器创建一个 loopback 环回接口，以便容器可以实现内部通信。实现过程中，同样为类型 libcontainer.Config 的 container 对象添加 Networks 属性，Network 的类型为 loopback，源码如下：

```
container.Networks = []*libcontainer.Network{
        {
             Mtu:     c.Network.Mtu,
             Address: fmt.Sprintf("%s/%d", "127.0.0.1", 0),
             Gateway: "localhost",
             Type:    "loopback",
        },
    }
```

至此，createNetwork 函数已经把与网络相关的配置，全部创建在类型为 libcontainer.Config 的 container 对象中，随时等候创建容器进程的来临。

7.5.2 调用 libcontainer 的 namespaces 启动容器

回到 execdriver.Run 函数，创建完 libcontainer.Config 实例 container，经过一系列处理之后，最终 execdriver 执行 namespaces.Exec 函数实现启动容器。启动过程中 container 对象依然是 namespace.Exec 函数中一个非常重要的参数。namespaces.Exec 代表着 execdriver 把启动

Docker 容器的工作交给 libcontainer，之后的程序执行将完全陷入 libcontainer。

调用 namespaces.Exec 的源码位于 ./docker/daemon/execdriver/native/driver.go#L102-L127，为了便于理解，简化之后如下：

```
namespaces.Exec(container, c.Stdin, c.Stdout, c.Stderr, c.Console, c.Rootfs,
dataPath, args, parameter_1, parameter_2)
```

其中 parameter_1 为定义的函数，如下所示：

```
func(container *libcontainer.Config, console, rootfs, dataPath, init string,
child *os.File, args []string) *exec.Cmd {
    c.Path = d.initPath
    c.Args = append([]string{
        DriverName,
        "-console", console,
        "-pipe", "3",
        "-root", filepath.Join(d.root, c.ID),
        "--",
    }, args...)

    // set this to nil so that when we set the clone flags anything else is reset
    c.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
    }
    c.ExtraFiles = []*os.File{child}

    c.Env = container.Env
    c.Dir = c.Rootfs

    return &c.Cmd
}
```

同样，parameter_2 也为定义的函数，如下所示：

```
func() {
    if startCallback != nil {
        c.ContainerPid = c.Process.Pid
        startCallback(c)
    }
}
```

parameter_1 以及 parameter_2 这两个函数均会在 libcontainer 的 namespaces 中发挥很大的作用，7.6 节将进行深入分析。

至此，execdriver 模块的执行部分已经结束，Docker Daemon 的运行陷入 libcontainer。

7.6　libcontainer 实现内核态网络配置

libcontainer 是一个 Linux 操作系统上容器技术的解决方案。libcontainer 指定了创建一个

容器时所需要的配置选项，同时它利用 Linux 中 namespaces 和 cgroups 等技术为使用者提供了一套 Golang 原生态的容器实现方案，并且没有使用任何外部依赖。用户借助 libcontainer，可以感受到众多操作命名空间、网络等资源的便利。

当 execdriver 调用 libcontainer 中 namespaces 包的 Exec 函数时，libcontainer 开始发挥其实现容器功能的作用。Exec 函数位于 ./docker/libcontainer/namespaces/exec.go#L24-L113。本节更多地关心 Docker 容器的网络创建，因此从这个角度来看 Exec 的实现可以分为三个步骤：

1）通过 createCommand 创建一个 Golang 语言内的 exec.Cmd 对象。

2）启动命令 exec.Cmd，创建容器内的第一个进程。

3）通过 InitializeNetworking 函数为容器进程初始化网络环境。

以下从源码实现的角度，详细分析这三个部分。

7.6.1 创建 exec.Cmd

提到 exec.Cmd，就不得不提 Go 语言标准库中的 os 包和 os/exec 包。前者提供了与平台无关的操作系统功能集，后者则提供了功能集里与命令执行相关的部分。

首先来看一下在 Go 语言中 exec.Cmd 的定义，如下所示：

```
type Cmd struct {
    Path string                          // 所需执行命令在系统中的路径
    Args []string                        // 传入命令的参数
    Env []string                         // 进程运行时的环境变量
    Dir string                           // 命令运行的工作目录
    Stdin io.Reader
    Stdout io.Writer
    Stderr io.Writer
    ExtraFiles []*os.File                // 进程所需打开的额外文件
    SysProcAttr *syscall.SysProcAttr     // 可选的操作系统属性
    Process *os.Process                  // 代表 Cmd 启动后，操作系统底层的具体进程
    ProcessState *os.ProcessState        // 进程退出后保留的信息
}
```

清楚 Cmd 的定义之后，再来分析 namespaces 包中的 Exec 函数，libcontainer 是如何来创建 exec.Cmd 的。在 Exec 函数的实现过程中，使用了以下源码实现 Exec.Cmd 的创建：

```
command := createCommand(container, console, rootfs, dataPath, os.Args[0],
syncPipe.Child(), args)
```

其中 createCommand 为 namespace.Exec 函数中传入的倒数第二个参数，类型为 CreateCommand。而 createCommand 只是 namespaces.Exec 函数的形参，真正的实参则为 execdriver 调用 namespaces.Exec 时的参数 parameter_1，源码如下：

```
func(container *libcontainer.Config, console, rootfs, dataPath, init string,
child *os.File, args []string) *exec.Cmd {
    c.Path = d.initPath
```

```
    c.Args = append([]string{
        DriverName,
        "-console", console,
        "-pipe", "3",
        "-root", filepath.Join(d.root, c.ID),
        "--",
    }, args...)

    // set this to nil so that when we set the clone flags anything else is reset
    c.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
    }
    c.ExtraFiles = []*os.File{child}

    c.Env = container.Env
    c.Dir = c.Rootfs

    return &c.Cmd
}
```

熟悉 exec.Cmd 的定义之后，分析以上源码就显得较为简单。为 Cmd 赋值的对象有 Path、Args、SysProcAttr、ExtraFiles、Env 和 Dir。其中需要特别注意的是 Path 的值 d.initPath，该路径下存放的是 dockerinit 的二进制文件，Docker 1.2.0 版本下，路径一般为“/var/lib/docker/init/dockerinit-1.2.0”。另外 SysProcAttr 使用以下的代码来赋值：

```
&syscall.SysProcAttr{
    Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
}
```

在 syscall.SysProAttr 对象中的 Cloneflags 属性中，即保留了 libcontainer.Config 类型的实例 container 中的 Namespaces 属性。换言之，通过 exec.Cmd 创建进程时，libcontainer 正是通过 Cloneflags 来实现在 Clone 系统调用中传入 namespaces 参数标志。

回到函数执行中，在函数的最后返回了 c.Cmd，命令创建完毕。

7.6.2　启动 exec.Cmd 创建进程

创建完 exec.Cmd，当然需要执行该命令，namespaces.Exec 函数中直接使用以下源码实现进程的启动：

```
if err := command.Start(); err != nil {
    return -1, err
}
```

这一部分的内容简单直接，Start() 函数用以完成指定命令 exec.Cmd 的启动执行，同时不等待其启动完便返回。Start() 函数的定义位于 os/exec 包。

进入 os/exec 包，查看 Start() 函数的实现，可以看到执行过程中，会对 command.Process

进行赋值，此时 command.Process 中会含有刚才启动进程的 PID 进程号，该 Pid 号在宿主机 pid namespace 下，而并非是新创建的 namespace 下的 Pid 号。

7.6.3 为容器进程初始化网络环境

上一环节实现了容器进程的启动，然而还没有为之配置相应的网络环境。namespaces.Exec 在之后的 InitializeNetworing 中实现了为容器进程初始化网络环境。初始化网络环境需要两个非常重要的参数：container 对象以及容器进程的 Pid 号。类型为 libcontainer.Config 的实例 container 包含用户对 Docker 容器的网络配置需求，另外容器进程的 Pid 可以使得创建的网络环境与进程新创建的 namespace 进行关联。

namespaces.Exec 中为容器进程初始化网络环境的代码实现位于 ./libcontainer/namespaces/exec.go#L75-L79，如下所示：

```
if err := InitializeNetworking(container, command.Process.Pid, syncPipe,
&networkState); err != nil {
    command.Process.Kill()
    command.Wait()
    return -1, err
}
```

InitializeNetworing 的作用很明显，即为创建的容器进程初始化网络环境。具体实现包含两个步骤：

1）先在容器进程的网络命名空间外部创建该容器所需的网络栈。

2）将创建的网络栈传递至容器的网络命名空间。

IntializeNetworking 的源代码实现位于 ./libcontainer/namespaces/exec.go#L176-L187，如下所示：

```
func InitializeNetworking(container *libcontainer.Config, nspid int, pipe
*syncpipe.SyncPipe, networkState *network.NetworkState) error {
    for _, config := range container.Networks {
        strategy, err := network.GetStrategy(config.Type)
        if err != nil {
            return err
        }
        if err := strategy.Create((*network.Network)(config), nspid,
        networkState); err != nil {
            return err
        }
    }
    return pipe.SendToChild(networkState)
}
```

以上源码实现过程，首先通过一个循环，遍历 libcontainer.Config 类型实例 container 中的网络属性 Networks；随后使用 GetStrategy 函数处理 Networks 中每一个对象的 Type 属性，

得出 Network 的类型，这里的类型有 3 种，分别为 loopback、veth 和 netns。loopback 环回接口针对 bridge 模式和 none 模式；veth 针对 bridge 模式；而 netns 针对 other container 模式。

得到 Network 的类型之后，libcontainer 创建相应的网络栈，具体实现使用每种网络栈类型下的 Create 函数。下面分析三种不同网络栈各自的创建流程。需要额外注意的是：容器网络栈的创建不在容器网络命名空间之内，而是在 Docker Daemon 所在的网络命名空间内。

1. loopback 网络栈的创建

loopback 是一种本地环回接口，libcontainer 创建 loopback 网络设备的实现代码位于 ./libcontainer/network/loopback.go#L13-L15，如下所示：

```
func (l *Loopback) Create(n *Network, nspid int, networkState *NetworkState) error {
    return nil
}
```

令人费解的是，libcontainer 在 loopback 接口的创建函数 Create 中，并没有实质性的内容，而是直接返回 nil。其实关于 loopback 接口的创建，要回到 Linux 内核为进程新建 net namespace 的阶段。当 libcontainer 执行 command.Start() 时，由于创建了一个新的 net namespace，故 Linux 内核会自动为新的 net namespace 创建一个 loopback 接口。当 Linux 内核创建完 loopback 接口之后，libcontainer 所做的工作即只保留 loopback 设备的默认配置，并在后续 libcontainer 的 Initialize 函数中实现启动该接口。

2. veth 网络栈的创建

veth 是 Docker 容器实际使用的网络策略之一，实现手段是：使用网桥 docker0 并创建 veth pair 虚拟网络接口对，最终使一个 veth 附加在宿主机的 docker0 网桥之上，而另一个 veth 安置在容器的 net namespace 内部。

libcontainer 中实现 veth 接口的代码非常通俗易懂，位于 ./docker/libcontainer/network/veth.go#L19-L50，如下所示：

```
name1, name2, err := createVethPair(prefix)
if err != nil {
    return err
}
if err := SetInterfaceMaster(name1, bridge); err != nil {
    return err
}
if err := SetMtu(name1, n.Mtu); err != nil {
    return err
}
if err := InterfaceUp(name1); err != nil {
    return err
}
if err := SetInterfaceInNamespacePid(name2, nspid); err != nil {
    return err
}
```

主要的流程包含以下四个步骤：

1）在宿主机上创建 veth pair。

2）将一个 veth 附加至 docker0 网桥上。

3）启动第一个 veth。

4）将第二个 veth 附加至 libcontainer 创建进程的 namespace 下（注意：并未启动第二个 veth）。

使用 Create 函数实现 veth pair 的创建之后，libcontainer 在 Initialize 函数中实现将网络 namespace 中的 veth 改名为“eth0”、设置网络设备的 MTU、以及启动网络接口等。

3. netns 网络栈的创建

netns 针对 Docker 容器的 other container 网络模式服务。netns 完成的工作是：将其他容器的 net namespace 路径，传递给需要创建 other container 网络模式的容器使用。

libcontainer 中实现 netns 策略的源码位于 ./docker/libcontainer/network/netns.go#L17-L20，如下所示：

```
func (v *NetNS) Create(n *Network, nspid int, networkState *NetworkState) error {
    networkState.NsPath = n.NsPath
    return nil
}
```

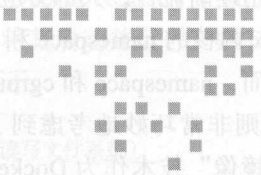
libcontainer 使用 Create 函数先将 NsPath 传递给新建容器，再在 Initialize 函数中实现将 net namespace 的文件描述符交由新创建容器使用，最终实现两个 Docker 容器共享同一个网络栈。

通过 Create 函数，Docker 容器相应的网络栈环境即已经完成创建，初始化工作有待 Initialize 函数来完成。详见本书第 13 章，有关 dockerinit 的执行将完成容器网络栈的初始化。

7.7 总结

如何使用 Docker 容器的网络，一直是工业界关心的问题。本章从 Linux 内核原理的角度阐述了什么是 Docker 容器，并对 Docker 容器的 4 种网络模式进行了初步的介绍，最终贯穿 Docker 架构中的多个模块，如 Docker Client、Docker Daemon、execdriver 以及 libcontainer，深入分析了 Docker 容器网络的实现。

目前，若只谈论 Docker，那么它还是只停留在单宿主机的场景上。如何面对跨宿主机的场景、如何实现分布式 Docker 容器的管理，目前为止还没有一个一劳永逸的解决方案。再者，一个解决方案的存在，总是会适应于一个应用场景。Docker 这种容器技术的发展，大大改善了传统模式下使用诸如虚拟机等传统计算单位存在的多种弊端，却在网络方面使得自身的使用存在瑕疵。希望本章是一个引子，介绍 Docker 容器网络，从源码的角度分析 Docker 容器网络之后，能使更多的 Docker 爱好者思考 Docker 容器网络的来龙去脉，并为 Docker 乃至容器技术的发展做出贡献。



第 8 章 Chapter 8

Docker 镜像

8.1 引言

2014 年，Docker 便在全球刮起了一阵又一阵的“容器风”，全球的开发者开始认识 Docker，学习 Docker。又过了一年，工业界对 Docker 的态度已经不再是了解与观望，转而是实践一波高过一波。Docker 目前的发展似乎并不会像其他昙花一现的技术一样，反而是凭借创新性的特性，Docker 在工业界经过实践与评估之后，显现了前所未有的潜力。

究其本质，“Docker 提供容器服务”这句话，相信很少有人会有异议。既然如此，Docker 提供的服务属于“容器”技术，那么反观“容器”技术的本质与发展历史，我们又可以发现什么呢？正如第 7 章所提到的，Docker 使用的“容器”技术，主要是以 Linux 内核的 namespace、cgroup 等特性为基础，保障进程或者进程组处于一个隔离、受限、安全的环境之中。Docker 第一个版本在 2013 年 3 月发行，而 cgroups 正式在 Linux 操作系统中的亮相可以追溯到 2007 年下半年，当时 cgroups 被合并至 Linux 内核 2.6.24 版本。这整整 6 年时间并不是 Linux 平台“容器”技术发展的真空期。2008 年 LXC（Linux Container）诞生，它简化了容器的创建与管理；之后业界一些 PaaS 平台也初步尝试采用容器技术作为其云应用的运行环境。而在 Docker 发布的同年，Google 也发布了开源容器管理工具 lmctfy。除此之外，若抛开 Linux 操作系统，其他操作系统（如 FreeBSD、Solaris 等）同样诞生了作用类似的“容器”技术，其发展历史更是可以追溯至千禧年初期。

总之，“容器”技术的发展不可谓短暂，然而论同时代的影响力，却鲜有 Docker 的媲美者。不论是云计算大潮催生了 Docker 技术，抑或是 Docker 技术赶上了云计算的大时代，毋庸置疑的是，Docker 作为技术领域的新宠儿，必将继续受到业界的广泛青睐。云计算时

代，分布式应用逐渐流行，大部分应用对自身的构建、交付与运行都有着与传统不一样的需求。借助 Linux 内核的 namespace 和 cgroup 等特性，自然可以实现应用运行环境的资源隔离与限制等；然而，namespace 和 cgroup 等内核特性却无法为容器的运行环境做全盘打包。而 Docker 的设计则非常巧妙地考虑到了这一点，除 namespace 和 cgroup 之外，Docker 另外采用了神奇的“镜像”技术作为 Docker 管理文件系统以及运行环境强有力的补充。Docker 灵活的“镜像”技术，在笔者看来，也是其大红大紫最重要的因素之一。

8.2 Docker 镜像介绍

Docker 诞生至今，很多容器领域从业者都会将 Docker 技术与虚拟机技术相提并论。提及 Docker 镜像，大家肯定也会联想到虚拟机中的镜像。镜像是一种文件存储形式，文件管理员可以通过技术手段将很多文件制作成一个镜像。对于虚拟机而言，镜像文件中存储着操作系统、文件系统内容、设备文件等。可以说，Docker 镜像与虚拟机镜像有很大的相似度，然而也有着本质的区别。相似的是，两者存储内容大致相同，都会含有文件系统内容；不同的是，Docker 镜像不含操作系统内容，同时 Docker 镜像由多个镜像组成。

根据 Docker 官方网站上的技术文档描述，image（镜像）是 Docker 术语的一种，对于容器而言，它代表一个只读的 layer。而 layer 则具体代表 Docker 容器文件系统中可叠加的一部分。

如此介绍 Docker 镜像，相信众多 Docker 爱好者理解起来仍然是云里雾里。那么理解之前，先让我们来认识一下与 Docker 镜像相关的 4 个概念：rootfs、union mount、image 以及 layer。

8.3 rootfs

rootfs 代表一个 Docker 容器在启动时（而非运行后）其内部进程可见的文件系统视角，或者 Docker 容器的根目录。当然，该目录下含有 Docker 容器所需要的系统文件、工具、容器文件等。

传统上，Linux 操作系统内核启动时，内核首先会挂载一个只读（read-only）的 rootfs，当系统检测其完整性之后，决定是否将其切换为读写（read-write）模式，或者最后在 rootfs 之上另行挂载一种文件系统并忽略 rootfs。Docker 架构下，依然沿用 Linux 中 rootfs 的思想。当 Docker Daemon 为 Docker 容器挂载 rootfs 的时候，与传统 Linux 内核类似，将其设定为只读模式。在 rootfs 挂载完毕之后，和 Linux 内核不一样的是，Docker Daemon 没有将 Docker 容器的文件系统设为读写模式，而是利用 Union Mount 的技术，在这个只读的 rootfs 之上再挂载一个读写的文件系统，挂载时该读写文件系统内空无一物。在这里，我们暂且把 Docker 容器的文件系统这么理解：只含只读的 rootfs 和可读写的文件系统。

举一个 Ubuntu 容器启动的例子。假设用户已经通过 Docker Registry 下拉了 Ubuntu:14.04 的镜像，并通过命令 docker run –it ubuntu:14.04 /bin/bash 将其启动并运行，则 Docker Daemon 为其创建的 rootfs 以及容器可读写的文件系统如图 8-1 所示。

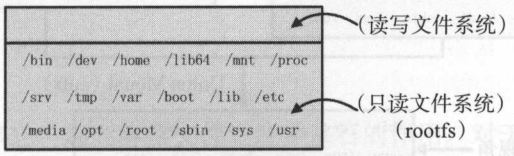


图 8-1　Ubuntu 14.04 容器文件系统示意图

顾名思义，该容器中的进程对 rootfs 中的内容只拥有读权限，对于读写文件系统中的内容既拥有读权限也拥有写权限。通过观察图 8-1 可以发现：容器虽然只有一个文件系统，但该文件系统由“两层”组成，分别为读写文件系统和只读文件系统。通过这样的理解，文件系统已然有了层级（layer）的意味。

简单来讲，可以将 Docker 容器的文件系统分为两部分，而前面提到的 Docker Daemon 利用 Union Mount 技术，将两者挂载。那么 Union Mount 又是一种怎样的技术？下一节将介绍 Union Mount 的概念。

8.4　Union Mount

Union Mount 代表一种文件系统挂载方式，允许同一时刻多种文件系统叠加挂载在一起，并以一种文件系统的形式，呈现多种文件系统内容合并后的目录。

一般情况下，若通过某种文件系统挂载内容至挂载点，挂载点目录中原先的内容将会被隐藏。而 Union Mount 则不会将挂载点目录中的内容隐藏，反而是将挂载点目录中的内容和被挂载的内容合并，并为合并后的内容提供一个统一独立的文件系统视角。通常来讲，被合并的文件系统中只有一个会以读写（read-write）模式挂载，其他文件系统的挂载模式均为只读（read-only）。实现这种 Union Mount 技术的文件系统一般称为联合文件系统（Union Filesystem），较为常见的有 UnionFS、aufs、OverlayFS 等。

Docker 实现容器文件系统 Union Mount 时，提供多种具体的文件系统解决方案，如 Docker 早期版本沿用至今的 AUFS，还有在 Docker 1.4.0 版本中开始支持的 OverlayFS 等。

为了更深入地了解 Union Mount，可以使用 aufs 文件系统来进一步阐述本章前面 Ubuntu:14.04 容器文件系统的例子，挂载 Ubuntu 14.04 文件系统的示意图如图 8-2 所示。

使用镜像 Ubuntu:14.04 创建的容器中，可以暂且将该容器的整个 rootfs 当成一个文件系统。前面也提到，挂载时读写文件系统中空无一物。既然如此，从用户视角来看，容器内文件系统和 rootfs 完全一样，用户完全可以按照往常习惯，无差别地使用自身视角下文件系统中的所有内容；然而，从内核的角度来看，两者有非常大的区别。追溯区别存在的根本原因，

那就不得不提及 aufs 等文件系统的 COW（copy-on-write）特性。

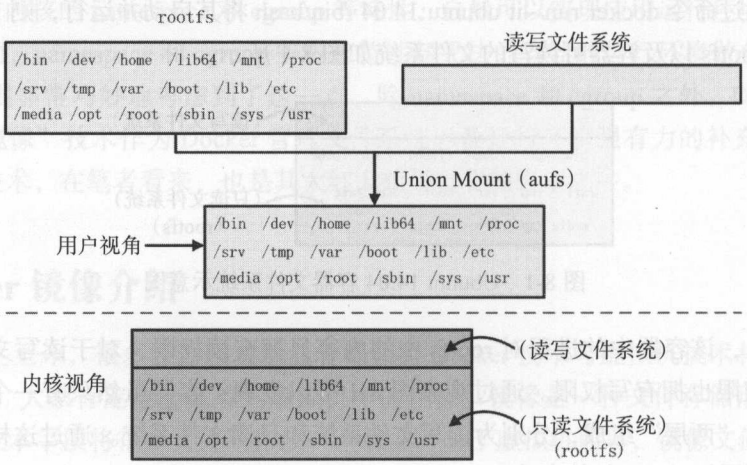


图 8-2 aufs 挂载 Ubuntu 14.04 文件系统的示意图

COW 文件系统和其他文件系统最大的区别就是：前者从不覆写已有文件系统中已有的内容。通过 COW 文件系统将两个文件系统（rootfs 和读写文件系统）合并，最终用户视角为合并后含有所有内容的文件系统，然而在 Linux 内核逻辑上依然可以区别两者，那就是用户对原先 rootfs 中的内容拥有只读权限，而对读写文件系统中的内容拥有读写权限。

既然对用户而言，全然不知哪些内容只读，哪些内容可读写，这些信息只有内核在接管，那么假设用户需要更新其视角下的文件 /etc/bash.bashrc，而该文件又恰巧是 rootfs 只读文件系统中的内容，内核是否会抛出异常或者驳回用户请求呢？答案是否定的。当此情形发生时，COW 文件系统首先不会覆写只读文件系统中的文件，即不会覆写 rootfs 中的 /etc/bash.bashrc，其次反而会将该文件复制至读写文件系统中，即将 /etc/bash.bashrc 复制至读写文件系统中的 /etc/bash.bashrc（此时，rootfs 文件系统和读写文件系统中各含有一份 /etc/bash.bashrc），最后再对后者进行更新操作。如此一来，纵使 rootfs 与读写文件系统中均有 /etc/bash.bashrc，诸如 aufs 类型的 COW 文件系统也能保证用户视角中只能看到读写文件系统中的 /etc/bash.bashrc，即更新后的内容。

当然，这样的特性同样支持 rootfs 中文件的删除等其他操作。例如：用户通过 apt-get 软件包管理工具安装 Golang，所有与 Golang 相关的内容都会安装在读写文件系统中，而不会安装在 rootfs 中。此时用户又希望通过 apt-get 软件包管理工具删除所有关于 MySQL 的内容，恰巧这部分内容又都存在于 rootfs 中，删除操作执行时同样不会删除 rootfs 实际存在的 MySQL，而是在读写文件系统中删除该部分内容，导致最终 rootfs 中的 MySQL 对容器用户不可见，也不可访。由于读写文件系统中根本不存在 MySQL 的相关内容，因此似乎在读写文件系统中找不到需要删除的对象。此时，aufs 保障在读写文件系统中对这些文件内容做相关的标记（whiteout），确保用户在查看文件系统内容时，读写文件系统中的 whiteout 将遮盖

住 rootfs 中的相应内容，导致这些内容不可见，以达到与删除这部分内容相类似的效果。

掌握 Docker 中 rootfs 以及 Union Mount 的概念之后，再来理解 Docker 镜像，就会有水到渠成的感觉。

8.5 image

Docker 中 rootfs 的概念，起到容器文件系统中基石的作用。对于容器而言，其只读的特性也是不难理解。神奇的是，实际情况下 Docker 架构中 rootfs 的设计与实现比前面的描述还要精妙得多。

继续以 Ubuntu 14.04 为例，虽然通过 aufs 可以实现 rootfs 与读写文件系统的合并，但是考虑到 rootfs 自身接近 200MB 的磁盘大小，如果以这个 rootfs 的粒度来实现容器的创建与迁移等，是否会稍显笨重？同时也会大大降低镜像的灵活性？而且，若用户希望拥有一个 Ubuntu 14.10 的 rootfs，那么是否有必要创建一个全新的 rootfs，毕竟 Ubuntu 14.10 和 Ubuntu 14.04 的 rootfs 中有很多一致的内容。

Docker 中 image 的概念，非常巧妙地解决了以上问题。最为简单地解释 image，它就是 Docker 容器中只读文件系统 rootfs 的一部分。换言之，实际上 Docker 容器的 rootfs 可以由多个 image 来构成。多个 image 构成 rootfs 的方式依然沿用 Union Mount 技术。

多个 image 构成的 rootfs 如图 8-3 所示（其中，rootfs 中每一层 image 中的内容划分只为了阐述清楚 rootfs 由多个 image 构成，并不代表实际情况下 rootfs 中的内容划分）：

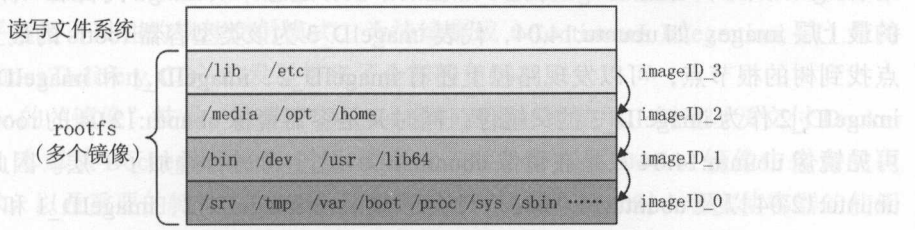


图 8-3　容器 rootfs 多 image 构成图

从图 8-3 中我们可以看出，容器 rootfs 包含 4 个 image，其中每个 image 中都有一些用户视角文件系统中的一部分内容。4 个 image 处于层叠的关系，除了最底层的 image，每一层的 image 都叠加在另一个 image 之上。另外，每一个 image 均含有一个 image ID，用于唯一地标记该 image。

基于以上概念，Docker Image 中又抽象出两种概念：父镜像以及基础镜像。除了容器 rootfs 最底层的镜像，其余镜像都依赖于其底下的一个或多个镜像。这种情况下，Docker 将下一层的镜像称为上一层镜像的父镜像。以图 9-3 为例，imageID_0 是 imageID_1 的父镜像，imageID_2 是 imageID_3 的父镜像，而 imageID_0 没有父镜像。对于最下层的镜像，即没有

父镜像的镜像，在 Docker 中我们习惯称之为基础镜像。

通过 image 的形式，原先较为臃肿的 rootfs 被逐渐打散成轻便的多层。除了轻便的特性之外，image 同时还有前面提到的只读特性，如此一来，在不同的容器、不同的 rootfs 中 image 完全可以用来复用。

多 image 组织关系与复用关系如图 8-4 所示（图中镜像名称的举例只为将 image 之间的关系阐述清楚，并不代表实际情况下相应名称 image 之间的关系）：

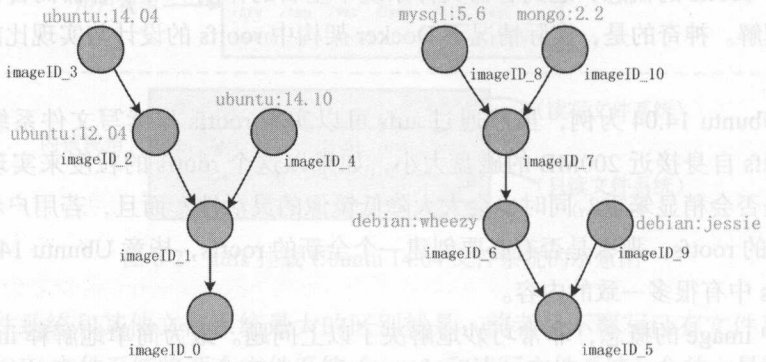


图 8-4　多 image 组织关系示意图

图 8-4 中，共罗列了 11 个 image，这 11 个 image 之间的关系呈现为一幅森林图。森林中含有两棵树，左边树中包含 5 个节点，即含有 5 个 image；右边树中包含 6 个节点，即含有 6 个 image。其中，有些 image 标记了加粗的字段，这意味该 image 代表某一种容器镜像 rootfs 的最上层 image。如 ubuntu:14.04，代表 imageID_3 为该类型容器 rootfs 的最上层，沿着该节点找到树的根节点，可以发现路径上还有 imageID_2、imageID_1 和 imageID_0。特殊的是，imageID_2 作为 imageID_3 的父镜像，同时又是容器镜像 ubuntu:12.04 的 rootfs 中的最上层，可见镜像 ubuntu:14.04 只是在镜像 ubuntu:12.04 之上再另行叠加了一层。因此，在下载镜像 ubuntu:12.04 以及 ubuntu:14.04 时，只会下载一份 imageID_2、imageID_1 和 imageID_0，实现 image 的复用。同时，右边树中 mysql:5.6、mongo:2.2、debian:wheezy 和 debian:jessie 也呈现同样的关系。

8.6　layer

在 Docker 中，术语 layer 是一个与 image 含义较为相近的词。容器镜像的 rootfs 是容器只读的文件系统，rootfs 又由多个只读的 image 构成。于是，rootfs 中每个只读的 image 都可以称为一个 layer。

除了只读的 image 之外，Docker Daemon 在创建容器时会在容器的 rootfs 之上，再挂载一层读写文件系统，而这一层文件系统也称为容器的一个 layer，常被称为 top layer。实际情

况下，Docker 还会在 rootfs 和 top layer 之间再挂载一个 layer，这一个 layer 中主要包含的内容是 /etc/hosts、/etc/hostname 以及 /etc/resolv.conf，一般这一个 layer 称为 init layer。为了简化阐述流程，我们暂不提 init layer。

因此，总之，Docker 容器中每一层只读的 image 以及最上层可读写的文件系统，均称为 layer。如此一来，layer 的范畴比 image 多了一层，即多包含了最上层的读写文件系统。

有了 layer 的概念，大家可以思考这样一个问题：容器文件系统分为只读的 rootfs，以及可读写的 top layer，那么容器运行时若在 top layer 中写入了内容，这些内容是否可以持久化，并且也被其他容器复用？

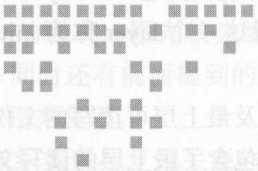
本章对于 image 的分析中，提到了 image 有复用的特性，既然如此，再提出一个更为大胆的假设：容器的 top layer 是否可以转变为 image？

答案是肯定的。Docker 的设计理念中，top layer 转变为 image 的行为（Docker 中称为 commit 操作）进一步释放了容器 rootfs 的灵活性。Docker 的开发者完全可以基于某个镜像创建容器做开发工作，并且无论在开发周期的哪个时间点，都可以对容器进行 commit，将所有 top layer 中的内容打包为一个 image，构成一个新的镜像。commit 完毕之后，用户完全可以基于新的镜像，进行开发、分发、测试、部署等。不仅 Docker commit 的原理如此，基于 Dockerfile 的 docker build 最为核心的思想，也是不断将容器的 top layer 转化为 image。

8.7　总结

Docker 风暴席卷全球并非偶然。如今的云计算时代下，轻量级容器技术与灵活的镜像技术相结合，似乎颠覆了以往的软件交付模式，为持续集成（Continuous Integration, CI）与持续交付（Continuous Delivery, CD）的发展带来了全新的契机。

理解 Docker 的“镜像”技术，有助于 Docker 爱好者更好地使用、创建以及交付 Docker 镜像。基于此，本章从 Docker 镜像的 4 个重要概念入手，介绍了 Docker 镜像中包含的内容，涉及的技术，以及重要的特性。Docker 引入优秀的“镜像”技术时，着实使容器的使用变得更为便利，也拓宽了 Docker 的使用范畴。然而，与此同时，我们也应该理性地看待镜像技术引入时，是否会带来其他副作用，如镜像技术是否会带来性能问题等都将是我们进一步思考的话题。



Chapter 9

第 9 章 Docker 镜像下载

9.1 引言

说 Docker Image 是 Docker 体系的价值所在，没有丝毫的夸张。Docker Image 作为容器运行环境的基石，彻底解放了 Docker 容器的生命力，也激发了用户对于容器运用的无限想象力。

玩转 Docker，必然离不开 Docker Image 的支持。然而“万物皆有源”，Docker Image 来自何方，Docker Image 又是通过何种途径传输到用户的宿主机，以致用户可以通过 Docker Image 创建容器的呢？回忆初次接触 Docker 的场景，大家肯定对两条命令不陌生：docker pull 和 docker run。这两条命令中，正是前者实现了 Docker Image 的下载。Docker Daemon 在执行这条命令时，会将 Docker Image 从 Docker Registry 下载至本地，并保存在本地 Docker Daemon 管理的 Graph 中。

谈及 Docker Registry，Docker 爱好者首先联想到的自然是 Docker Hub。Docker Hub 作为 Docker 官方支持的 Docker Registry，拥有全球成千上万的 Docker Image。全球的 Docker 爱好者除了可以下载 Docker Hub 开放的镜像资源之外，还可以向 Docker Hub 贡献镜像资源。在 Docker Hub 上，用户不仅可以享受公有镜像带来的便利，而且可以创建私有镜像库。Docker Hub 是全国最大的 Public Registry，另外 Docker 还支持用户自定义创建 Private Registry。Private Registry 主要的功能是为私有网络提供 Docker 镜像的专属服务，一般而言，镜像种类适应用户需求，私密性较高，且不会占用公有网络带宽。

本章主要从 Docker 1.2.0 源码的角度分析 Docker 下载 Docker Image 的过程。分析内容主要包括以下 4 部分：

1）概述 Docker 镜像下载的流程，涉及 Docker Client、Docker Server 以及 Docker Daemon；

2）Docker Client 处理并发送 docker pull 请求；

3）Docker Server 接收 docker pull 请求，创建镜像下载任务并触发执行；

4）Docker Daemon 执行镜像下载任务，并存储镜像至 Graph。

9.2　Docker 镜像下载流程

Docker Image 作为 Docker 生态中的精髓，下载过程中需要 Docker 架构中多个组件的协作，下载流程如图 9-1 所示。

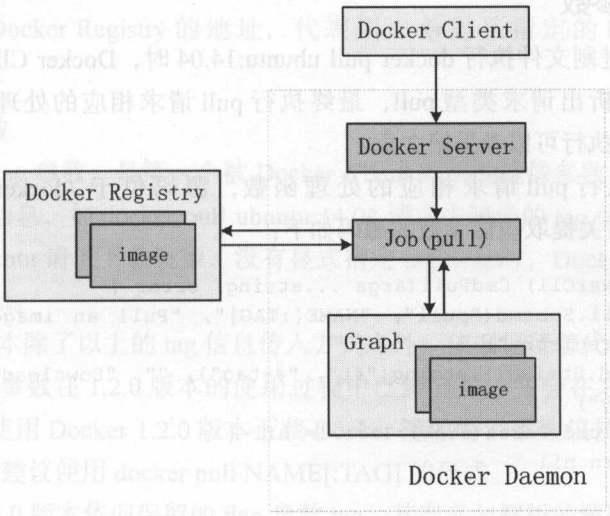


图 9-1　Docker 镜像下载流程图

如图 9-1 所示，Docker Image 的下载流程可以归纳为以下 3 个步骤：

1）用户通过 Docker Client 发送 pull 请求，用于让 Docker Daemon 下载指定名称的镜像；

2）Docker Server 接收 Docker 镜像的 pull 请求，创建下载镜像任务并触发执行；

3）Docker Daemon 执行镜像下载任务，从 Docker Registry 中下载指定镜像，并将其存储于本地的 Graph 中。

9.3　Docker Client

Docker 架构中，Docker 用户的角色绝大多数由 Docker Client 来扮演。因此，用户对

Docker 的管理请求全部由 Docker Client 来发送，Docker 镜像下载请求自然也不例外。

为了更清晰地描述 Docker 镜像下载，本节结合具体的命令进行分析，命令如下：

```
docker pull ubuntu:14.04
```

此命令的含义是：通过 docker 二进制可执行文件，执行镜像下载的 pull 命令，镜像参数为 ubuntu:14.04，镜像名称为 ubuntu，镜像标签（tag）为 14.04。此命令一经发起，第一个接受并处理的 Docker 组件为 Docker Client，执行内容包括以下三个步骤：

1）解析命令中与 Docker 镜像相关的参数；

2）配置 Docker 下载镜像时所需的认证信息；

3）发送 RESTful 请求至 Docker Daemon。

9.3.1 解析镜像参数

通过 docker 二进制文件执行 docker pull ubuntu:14.04 时，Docker Client 首先会创建，随后通过参数处理分析出请求类型 pull，最终执行 pull 请求相应的处理函数。关于 Docker Client 的创建与命令执行可以参见第 2 章。

Docker Client 执行 pull 请求相应的处理函数，源码位于 ./docker/api/client/command.go#L1183-L1244，有关提取镜像参数的源码如下：

```
func (cli *DockerCli) CmdPull(args ...string) error {
    cmd := cli.Subcmd("pull", "NAME[:TAG]", "Pull an image or a repository
    from the registry")
    tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a
    repository")
    if err := cmd.Parse(args); err != nil {
        return nil
    }

    if cmd.NArg() != 1 {
        cmd.Usage()
        return nil
    }
    var (
        v       = url.Values{}
        remote = cmd.Arg(0)
    )

    v.Set("fromImage", remote)

    if *tag == "" {
        v.Set("tag", *tag)
    }

    remote, _ = parsers.ParseRepositoryTag(remote)
    // Resolve the Repository name from fqn to hostname + name
```

```
    hostname, _, err := registry.ResolveRepositoryName(remote)
    if err != nil {
        return err
    }
    ...
}
```

结合命令 docker pull ubuntu:14.04 来分析 CmdPull 函数的定义，我们可以发现，该函数传入的形参为 args，实参是字符串 ubuntu:14.04。另外，纵观以上源码，可以发现 Docker Client 解析的镜像参数无外乎 4 个：tag、remote、v 以及 hostname，四者各自的作用如下。

- tag：带有 Docker 镜像的标签（已弃用）；
- remote：带有 Docker 镜像的名称与标签；
- v：类型为 url.Values，实质是一个 map 类型，用于配置请求中 URL 的查询参数；
- hostname：Docker Registry 的地址，代表用户希望从指定的 Docker Registry 下载 Docker 镜像。

1. 解析 tag 参数

Docker 镜像的 tag 参数，是第一个被 Docker Client 解析的镜像参数，代表用户所需下载 Docker 镜像的标签信息，如 docker pull ubuntu:14.04 请求中镜像的 tag 信息为 14.04，若用户使用 docker pull ubuntu 请求下载镜像，没有显式指定 tag 信息时，Docker Client 会默认该镜像的 tag 信息为 latest。

Docker 1.2.0 版本除了以上的 tag 信息传入方式之外，依旧保留着代表镜像标签的 flag 参数 tag，而这个 flag 参数在 1.2.0 版本的使用过程中已经弃用，并会在之后新版本的 Docker 中被移除，因此在使用 Docker 1.2.0 版本下载 Docker 镜像时，不建议使用 flag 参数 tag。传入 tag 信息的方式，建议使用 docker pull NAME[:TAG] 的形式。

关于 Docker 1.2.0 版本依旧保留的 flag 参数 tag，其定义与解析的源码位于：./docker/api/client/commands.go#1185-L1188，如下：

```
tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a
repository")
if err := cmd.Parse(args); err != nil {
    return nil
}
```

以上源码说明：CmdPull 函数解析 tag 参数时，Docker Client 首先定义一个 flag 参数，flag 参数的名称为“#t”或者“#-tag”，用途为：指定 Docker 镜像的 tag 参数，默认值为空字符串；随后通过 cmd.Parse(args) 的执行，解析 args 中的 tag 参数。

2. 解析 remote 参数

Docker Client 解析完 tag 参数之后，同样需要解析出 Docker 镜像所属的 repository，如请求 docker pull ubuntu:14.04 中，Docker 镜像为 ubuntu:14.04，镜像的 repository 信息为

ubuntu，镜像的 tag 信息为 14.04。

Docker Client 通过解析 remote 参数，使得 remote 参数携带 repository 信息和 tag 信息。Docker Client 解析 remote 参数的第一个步骤，源码如下：

```
remote = cmd.Arg(0)
```

其中，cmd 的第一个参数赋值给 remote，以 docker pull ubuntu:14.04 为例，cmd.Arg(0) 为 ubuntu:14.04，则赋值后 remote 值为 ubuntu:14.04。此时 remote 参数不仅包含 Docker 镜像的 repository 信息，还包含 tag 信息。若用户请求中带有 Docker Registry 的信息，如 docker pull localhost.localdomain:5000/docker/ubuntu:14.04，cmd.Arg(0) 为 localhost.localdomain:5000/docker/ubuntu:14.04，则赋值后 remote 值为 localhost.localdomain:5000/docker/ubuntu:14.04，此时 remote 参数同时包含 Docker Registry 信息、repository 信息以及 tag 信息。

随后，在解析 remote 参数的第二个步骤中，Docker Client 通过解析赋值完毕的 remote 参数解析出 repository 信息，并再次覆写 remote 参数的值，源码如下：

```
remote, _ = parsers.ParseRepositoryTag(remote)
```

ParseRepositoryTag 的作用是：解析出 remote 参数的 repository 信息和 tag 信息，该函数的实现位于 ./docker/pkg/parsers/parsers.go#L72-L81，源码如下：

```
func ParseRepositoryTag(repos string) (string, string) {
    n := strings.LastIndex(repos, ":")
    if n < 0 {
        return repos, ""
    }
    if tag := repos[n+1:]; !strings.Contains(tag, "/") {
        return repos[:n], tag
    }
    return repos, ""
}
```

以上函数的实现过程充分考虑了多种不同 Docker Registry 的情况，如：请求 docker pull ubuntu:14.04 中 remote 参数为 ubuntu:14.04，而请求 docker pull localhost.localdomain:5000/docker/ubuntu:14.04 中用户指定了 Docker Registry 的地址 localhost.localdomain:5000/docker，故 remote 参数还携带了 Docker Registry 信息。

ParseRepositoryTag 函数首先从 repos 参数的尾部往前寻找“:”，若不存在，则说明用户没有显式指定 Docker 镜像的 tag，返回整个 repos 作为 Docker 镜像的 repository；若“:”存在，则说明用户显式指定了 Docker 镜像的 tag，“:”前的内容作为 repository 信息，“:”后的内容作为 tag 信息，并返回两者。

ParseRepositoryTag 函数执行完，回到 CmdPull 函数，返回内容的 repository 信息将覆写 remote 参数。对于请求 docker pull localhost.localdomain:5000/docker/ubuntu:14.04，remote 参数被覆写后，值为 localhost.localdomain:5000/docker/ubuntu，携带 Docker Registry 信息以及

repository 信息。

3. 配置 url.Values

Docker Client 发送请求给 Docker Server 时，需要为请求配置 URL 的查询参数。CmdPull 函数的执行过程中创建 url.Value 并配置的源码实现位于 ./docker/api/client/commands.go#L1194-L1203，如下：

```
var (
    v      = url.Values{}
    remote = cmd.Arg(0)
)

v.Set("fromImage", remote)

if *tag == "" {
    v.Set("tag", *tag)
}
```

其中，变量 v 的类型是 url.Values，最终为 URL 配置的查询参数有两个，分别为“fromImage”与“tag”，“fromImage”的值是 remote 参数没有被覆写时的值，“tag”的值一般为空，原因是 tag 参数已弃用，一般不使用 flag 参数 tag。

4. 解析 hostname 参数

Docker Client 解析镜像参数时，还有一个重要的环节，那就是解析 Docker Registry 的地址信息。若用户希望从指定的 Docker Registry 中下载 Docker 镜像，则 Docker Client 需要考虑这种情况，为用户解析出 Docker Registry 的地址。

解析 Docker Registry 地址的代码实现位于 ./docker/api/client/commands.go#L1207，如下：

```
hostname, _, err := registry.ResolveRepositoryName(remote)
```

Docker Client 通过包 registry 中的函数 ResolveRepositoryName 来解析 hostname 参数，传入的实参为 remote，即去 tag 化的 remote 参数。ResolveRepositoryName 函数的源码实现位于 ./docker/registry/registry.go#L237-L259，如下：

```
func ResolveRepositoryName(reposName string) (string, string, error) {
    if strings.Contains(reposName, "://") {
        // It cannot contain a scheme!
        return "", "", ErrInvalidRepositoryName
    }
    nameParts := strings.SplitN(reposName, "/", 2)
    if len(nameParts) == 1
    || (!strings.Contains(nameParts[0], ".") && !strings.Contains(nameParts[0], ":") &&
    nameParts[0] != "localhost") {
        // This is a Docker Index repos (ex: samalba/hipache or ubuntu)
        err := validateRepositoryName(reposName)
        return IndexServerAddress(), reposName, err
```

```
    }
    hostname := nameParts[0]
    reposName = nameParts[1]
    if strings.Contains(hostname, "index.docker.io") {
        return "", "", fmt.Errorf("Invalid repository name, try \"%s\"
        instead", reposName)
    }
    if err := validateRepositoryName(reposName); err != nil {
        return "", "", err
    }

        return hostname, reposName, nil
}
```

ResolveRepositoryName 函数首先通过“/”分割字符串 reposName，如下：

```
nameParts := strings.SplitN(reposName, "/", 2)
```

如果 nameParts 的长度为 1，则说明 reposName 中不含有字符“/”，这意味着用户没有指定 Docker Registry。另外，形如“samalba/hipache”的 reposName 同样说明用户并没有指定 Docker Registry，其中 samalba 为用户在 Docker Hub 上的用户名。当用户没有指定 Docker Registry 时，Docker Client 默认返回 IndexServerAddress()，该函数返回常量 INDEXSERVER，值为“https://index.docker.io/v1”。也就是说，当用户下载 Docker 镜像时，若不指定 Docker Registry，默认情况下，Docker Client 通知 Docker Daemon 从 Docker Hub 上下载镜像。例如：请求 docker pull ubuntu:14.04，由于没有指定 Docker Registry，Docker Client 默认使用全球最大的 Docker Registry——Docker Hub。

当不满足返回默认 Docker Registry 时，Docker Client 通过解析 reposNames，得出用户指定的 Docker Registry 地址。例如：请求 docker pull localhost.localdomain:5000/docker/ubuntu:14.04 中，解析出的 Docker Registry 地址为 localhost.localdomain:5000。

至此，与 Docker 镜像相关的参数已经全部解析完毕，Docker Client 将携带这部分重要信息以及用户的认证信息来构建 RESTful 请求，发送给 Docker Server。

9.3.2 配置认证信息

用户下载 Docker 镜像时，Docker 同样支持用户信息的认证。用户认证信息由 Docker Client 配置；Docker Client 发送请求至 Docker Server 时，用户认证信息也被一并发送；随后，Docker Daemon 处理下载 Docker 镜像的请求时，用户认证信息将在 Docker Registry 中验证。

Docker Client 配置用户认证信息包含两个步骤，实现源码如下：

```
cli.LoadConfigFile()
// Resolve the Auth config relevant for this server
authConfig := cli.configFile.ResolveAuthConfig(hostname)
```

可见，第一个步骤是使 cli（Docker Client）加载 ConfigFile，ConfigFile 是 Docker Client

用于存放用户在 Docker Registry 上认证信息的对象。DockerCli、ConfigFile 以及 AuthConfig 三种数据结构之间的关系如图 9-2 所示。

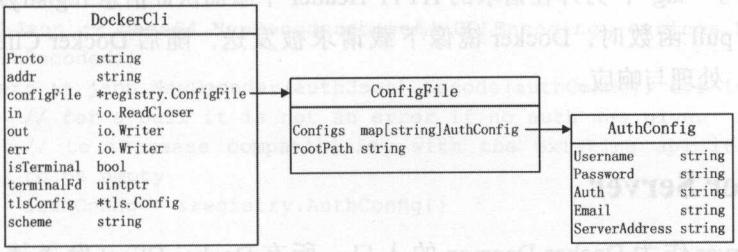


图 9-2　DockerCli、ConfigFile 以及 AuthConfig 关系图

DockerCli 结构体的属性 configFile 是一个指向 registry.ConfigFile 的指针，而 ConfigFile 结构体的属性 Configs 属于 map 类型，其中 key 为 string，代表 Docker Registry 的地址，value 的类型为 AuthConfig。AuthConfig 类型的具体含义为用户在某个 Docker Registry 上的认证信息，包含用户名、密码、认证信息、邮箱地址以及服务器地址等。

加载完用户所有的认证信息之后，Docker Client 配置用户认证信息的第二个步骤是：通过用户指定的 Docker Registry，即之前解析出的 hostname 参数，从用户所有的认证信息中找出与指定 hostname 相匹配的认证信息。新创建的 authConfig，类型即为 AuthConfig，将会作为用户在指定 Docker Registry 上的认证信息，发送至 Docker Server。

9.3.3　发送 API 请求

解析完所有的 Docker 镜像参数，并且配置完毕用户的认证信息之后，Docker Client 需要使用这些信息正式发送镜像下载的请求至 Docker Server。

Docker Client 定义了 pull 函数来实现发送镜像下载请求至 Docker Server，源码位于 ./docker/api/client/commands.go#L1217-L1229，如下：

```
pull := func(authConfig registry.AuthConfig) error {
    buf, err := json.Marshal(authConfig)
    if err != nil {
        return err
    }
    registryAuthHeader := []string{
        base64.URLEncoding.EncodeToString(buf),
    }

    return cli.stream("POST", "/images/create?"+v.Encode(), nil, cli.out,
    map[string][]string{
        "X-Registry-Auth": registryAuthHeader,
    })
}
```

pull 函数的实现较为简单，首先通过 authConfig 对象创建 registryAuthHeader，最后发送 POST 请求，请求的 URL 为“/images/create?”+v.Encode()，在 URL 中传入查询参数，包括“fromImage”与“tag”，另外在请求的 HTTP Header 中添加认证信息 registryAuthHeader。

执行以上 pull 函数时，Docker 镜像下载请求被发送，随后 Docker Client 等待 Docker Server 的接收、处理与响应。

9.4 Docker Server

Docker Server 作为 Docker Daemon 的入口，所有 Docker Client 发送请求都由 Docker Server 接收。Docker Server 通过解析请求的 URL 与请求方法，最终路由分发至相应的处理方法来处理。Docker Server 的创建与请求处理可以参见第 5 章。

Docker Server 接收到镜像下载请求之后，通过路由分发最终由具体的处理方法——postImagesCreate 来处理。postImagesCreate 的源码实现位于 ./docker/api/server/server.go#L466-L524，其执行流程主要分为 3 部分：

1）解析 HTTP 请求中包含的请求参数，包括 URL 中的查询参数、HTTP Header 中的认证信息等；

2）创建镜像下载 Job，并为该 Job 配置环境变量；

3）触发执行镜像下载 Job。

9.4.1 解析请求参数

Docker Server 接收到 Docker Client 发送的镜像下载请求之后，首先解析请求参数，并为后续 Job 的创建与运行提供参数依据。Docker Server 解析的请求参数主要有 HTTP 请求 URL 中的查询参数“fromImage”、“repo”以及“tag”，还有 HTTP 请求 Header 中的“X-Registry-Auth”。

请求参数解析的源码如下：

```
var (
    image = r.Form.Get("fromImage")
    repo  = r.Form.Get("repo")
    tag   = r.Form.Get("tag")
    job   *engine.Job
)
authEncoded := r.Header.Get("X-Registry-Auth")
```

需要特别说明的是，通过“fromImage”解析出的 image 变量包含镜像 repository 名称与镜像 tag 信息。例如，若用户请求为 docker pull ubuntu:14.04，则通过“fromImage”解析出的 image 变量值为 ubuntu:14.04，并非只有 Docker 镜像的名称。

另外，Docker Server 通过从 HTTP Header 中解析出 authEncoded，还原出类型为 registry.

AuthConfig 的对象 authConfig，源码实现如下：

```
authConfig := &registry.AuthConfig{}
if authEncoded != "" {
    authJson := base64.NewDecoder(base64.URLEncoding, strings.NewReader
    (authEncoded))
    if err := json.NewDecoder(authJson).Decode(authConfig); err != nil {
        // for a pull it is not an error if no auth was given
        // to increase compatibility with the existing api it is defaulting
        to be empty
        authConfig = &registry.AuthConfig{}
    }
}
```

解析出 HTTP 请求中的参数之后，Docker Server 对于 image 参数再次进行解析，从中解析出属于镜像的 repository 与 tag 信息，其中 repository 有可能暂时包含 Docker Registry 信息，源码实现如下：

```
if tag == "" {
    image, tag = parsers.ParseRepositoryTag(image)
}
```

Docker Server 的参数解析工作至此全部完成，之后 Docker Server 将创建镜像下载任务并开始执行。

9.4.2　创建并配置 Job

Docker Server 只负责接收 Docker Client 发送的请求，并将其路由分发至相应的处理方法来处理，最终的请求执行还需要 Docker Daemon 来协作完成。Docker Server 在处理方法中，通过创建 Job 并触发 job 执行的形式，把控制权交给 Docker Daemon。

Docker Server 创建镜像下载 job 并配置环境变量的源码实现如下：

```
job = eng.Job("pull", image, tag)
job.SetenvBool("parallel", version.GreaterThan("1.3"))
job.SetenvJson("metaHeaders", metaHeaders)
job.SetenvJson("authConfig", authConfig)
```

其中，创建的 Job 名为 pull，含义是下载 Docker 镜像，传入参数为 image 与 tag，配置的环境变量有 parallel、metaHeaders 与 authConfig。

9.4.3　触发执行 Job

Docker Server 创建完 Docker 镜像下载 Job 之后，需要触发该 Job 执行，实现将控制权交给 Docker Daemon。

Docker Server 触发执行 Job 的源码如下：

```
if err := job.Run(); err != nil {
    if !job.Stdout.Used() {
        return err
    }
    sf := utils.NewStreamFormatter(version.GreaterThan("1.0"))
    w.Write(sf.FormatError(err))
}
```

由于 Docker Daemon 在启动时，已经配置了名为“pull”的 Job 所对应的处理方法，实际为 graph 包中的 CmdPull 函数，故一旦该 Job 被触发执行，控制权将直接交给 Docker Daemon 的 CmdPull 函数。Docker Daemon 启动时 Engine 的处理方法注册可以参见第 3 章。

9.5 Docker Daemon

Docker Daemon 是完成 Job 执行的主要载体。Docker Server 为镜像下载 Job 准备好所有的参数配置之后，只等 Docker Daemon 来完成执行，并返回相应的信息，Docker Server 再将响应信息返回至 Docker Client。Docker Daemon 对于镜像下载 Job 的执行涉及的内容较多：首先解析 Job 参数，获取 Docker 镜像的 repository、tag 以及 Docker Registry 信息等；随后与 Docker Registry 建立会话（session）；然后通过会话下载 Docker 镜像；接着将 Docker 镜像下载至本地并存储于 Graph 中；最后在 TagStore 中标记该镜像。

Docker Daemon 对于镜像下载 Job 的执行主要依靠 CmdPull 函数。这个 CmdPull 函数与 Docker Client 的 CmdPull 函数完全不同，前者是为了代替用户发送镜像下载的请求至 Docker Daemon，而 Docker Daemon 的 CmdPull 函数则实现代替用户真正完成镜像下载的任务。调用 CmdPull 函数的对象类型为 TagStore，其源码实现位于 ./docker/graph/pull.go。

9.5.1 解析 Job 参数

正如 Docker Client 与 Docker Server 一样，Docker Daemon 执行镜像下载 Job 时的第一个步骤也是解析参数。解析工作一方面要确保传入参数无误，另一方面也要按需为 Job 提供参数依据。表 9-1 罗列出 Docker Daemon 解析的 Job 参数。

表 9-1 Docker Daemon 解析的 Job 参数

参数名称	参数描述
localName	代表镜像的 repository 信息，有可能携带 Docker Registry 信息
tag	代表镜像的标签信息，默认为 latest
authConfig	代表用户在指定 Docker Registry 上的认证信息
metaHeaders	代表请求中的 HTTP Header 信息
hostname	代表 Docker Registry 信息，从 localName 解析获得，默认为 Docker Hub 地址
remoteName	代表 Docker 镜像的 repository 名称信息，不携带 Docker Registry 信息
endpoint	代表 Docker Registry 完整的 URL，从 hostname 扩展获得

参数解析过程中，Docker Daemon 还添加了一些精妙的设计。如：在 TagStore 类型中设计了 pullingPool 对象，用于保存正在下载的 Docker 镜像，下载完毕之前禁止其他 Docker Client 发起相同镜像的下载请求，下载完毕之后 pullingPool 中的该记录被清除。Docker Daemon 一旦解析出 localName 与 tag 两个参数信息，就立即检测 pullingPool，源码实现位于 ./docker/graph/pull.go#L36-L46，如下：

```
c, err := s.poolAdd("pull", localName+":"+tag)
if err != nil {
    if c != nil {
        // Another pull of the same repository is already taking place; just
        wait for it to finish
        job.Stdout.Write(sf.FormatStatus("", "Repository %s already being
        pulled by another client. Waiting.", localName))
        <-c
        return engine.StatusOK
    }
    return job.Error(err)
}
defer s.poolRemove("pull", localName+":"+tag)
```

9.5.2　创建 session 对象

为了下载 Docker 镜像，Docker Daemon 与 Docker Registry 需要建立通信。为了保障两者之间通信的可靠性，Docker Daemon 采用了 session 机制。Docker Daemon 每收到一个 Docker Client 的镜像下载请求，都会创建一个与之相应的 Docker Registry 的 session，之后所有的网络数据传输都在该 session 上完成。包 registry 定义了 session，位于 ./docker/registry/registry.go，如下：

```
type Session struct {
    authConfig     *AuthConfig
    reqFactory     *utils.HTTPRequestFactory
    indexEndpoint string
    jar            *cookiejar.Jar
    timeout        TimeoutType
}
```

CmdPull 函数中创建 session 的源码实现如下：

```
r, err := registry.NewSession(authConfig, registry.HTTPRequestFactory(metaHeade
rs), endpoint, true)
```

创建的 session 对象为 r，在执行镜像下载过程中，多数与镜像相关的数据传输均在 r 这个 seesion 的基础上完成。

9.5.3 执行镜像下载

Docker Daemon 之前所有的操作都属于配置阶段，从解析 Job 参数，到建立 session 对象，而并未与 Docker Registry 建立实际的连接，并且也还未真正传输过有关 Docker 镜像的内容。

完成所有的配置之后，Docker Daemon 进入 Docker 镜像下载环节，实现 Docker 镜像的下载，源码位于 ./docker/graph/pull.go#L69-L71，如下：

```
if err = s.pullRepository(r, job.Stdout, localName, remoteName, tag, sf, job.
GetenvBool("parallel")); err != nil {
    return job.Error(err)
}
```

以上代码中 pullRepository 函数包含了镜像下载整个流程的林林总总，下载流程如图 9-3 所示。

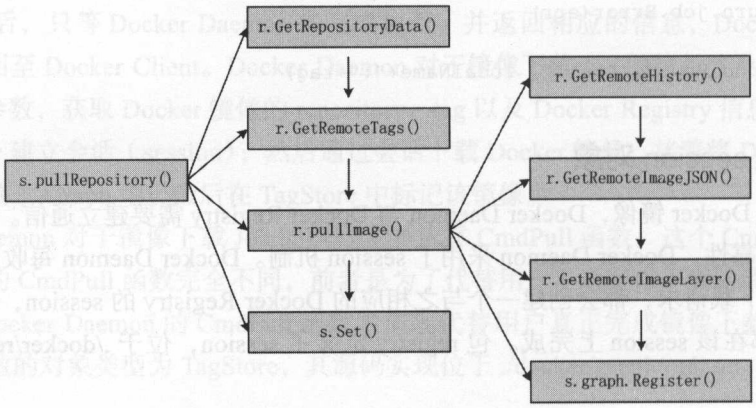


图 9-3 pullRepository 中镜像下载的流程图

图 9-3 中各个环节的简要功能介绍见表 9-2。

表 9-2 pullRepository 各环节功能

函数名称	功能介绍
r.GetRepositoryData()	获取指定 repository 中所有镜像的 ID 信息
r.GetRemoteTags()	获取指定 repository 中所有的 tag 信息
r.pullImage()	从 Docker Registry 下载 Docker 镜像
r.GetRemoteHistory()	获取指定 image 所有祖先镜像的 ID 信息
r.GetRemoteImageJSON()	获取指定 image 的 json 信息
r.GetRemoteImageLayer()	获取指定 image 的 layer 信息
s.graph.Register()	将下载的镜像在 TagStore 的 graph 中注册
s.Set()	在 TagStore 中添加新下载的镜像信息

分析 pullRepository 的整个流程之前，我们有必要了解 pullRepository 函数调用者的类型 TagStore。TagStore 是 Docker 镜像方面涵盖内容最多的数据结构：一方面 TagStore 管理 Docker 的 Graph，另一方面 TagStore 还管理 Docker 的 repository 记录。除此之外，TagStore 还管理 4.6.7 节提到的对象 pullingPool 以及 pushingPool，保证 Docker Daemon 在同一时刻只为一个 Docker Client 执行同一镜像的下载或上传。TagStore 结构体的定义位于 ./docker/graph/tags.go#L20-L29，如下：

```
type TagStore struct {
    path         string
    graph        *Graph
    Repositories map[string]Repository
    sync.Mutex
    // FIXME: move push/pull-related fields
    // to a helper type
    pullingPool map[string]chan struct{}
    pushingPool map[string]chan struct{}
}
```

下面将重点分析 pullRepository 的整个流程。

1. GetRepositoryData

使用 Docker 下载镜像时，用户往往指定的是 Docker 镜像的名称，如：请求 docker pull ubuntu:14.04 中镜像名称为 ubuntu。GetRepositoryData 的作用则是获取镜像名称所在 repository 中所有 image 的 ID 信息。

GetRepositoryData 的源码实现位于 ./docker/registry/session.go#L255-L324。获取 repository 中 image 的 id 信息的目标 URL 地址的源码如下：

```
repositoryTarget := fmt.Sprintf("%srepositories/%s/images", indexEp, remote)
```

因此，docker pull ubuntu:14.04 请求被执行时，repository 的目标 URL 地址为 https://index.docker.io/v1/repositories/ubuntu/images，访问该 URL 可以获得有关 ubuntu 这个 repository 中所有 image 的 ID 信息，部分 image 的 ID 信息如下：

```
[{"checksum": "", "id": "2427658c75a1e3d0af0e7272317a8abfaee4c15729b6840e3c2fc
a342fe47bf1"}, {"checksum": "", "id": "81fbd8fa918a14f4ebad9728df6785c53721827
9081c7a120d72399d3a5c94a5"}, {"checksum": "", "id": "ec69e8fd6b0236b67227869b6
d6d119f033221dd0f01e0f569518edabef3b72c"}, {"checksum": "", "id": "9e8dc15b6d3
27eaac00e37de743865f45bee3e0ae763791a34b61e206dd5222e"}, {"checksum": "", "id":
"78949b1e1cfdcd5db413c300023b178fc4b59c0e417221c0eb2ffbbd1a4725cc"},……]
```

获取以上信息之后，Docker Daemon 通过 RepositoryData 和 ImgData 类型对象来存储 ubuntu 这个 repository 中所有 image 的信息，RepositoryData 和 ImgData 中数据结构的关系如图 9-4 所示。

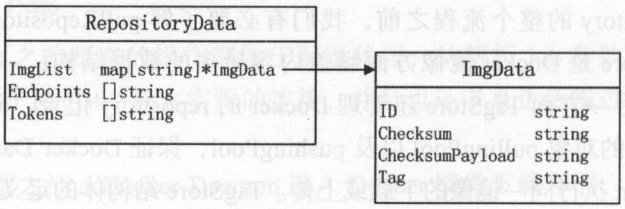


图 9-4 RepositoryData 和 ImgData 中数据结构的关系

GetRepositoryData 执行过程中，会为指定 repository 中的每一个 image 创建一个 ImgData 对象，并最终将所有 ImgData 存放在 RepositoryData 的 ImgList 属性中，ImgList 的类型为 map，key 为 image 的 ID，value 指向 ImgData 对象。此时 ImgData 对象中只有属性 ID 与 Checksum 有内容。

2. GetRemoteTags

使用 Docker 下载镜像时，用户除了指定 Docker 镜像的名称之外，一般还需要指定 Docker 镜像的 tag，如：请求 docker pull ubuntu:14.04 中镜像名称为 ubuntu，镜像 tag 为 14.04，假设用户不显式指定 tag，则默认 tag 为 latest。GetRemoteTags 的作用则是获取镜像名称所在 repository 中所有 tag 的信息。

GetRemoteTags 的源码实现位于 ./docker/registry/session.go#L195-234。获取 repository 中所有 tag 信息的目标 URL 地址的源码如下：

```
endpoint := fmt.Sprintf("%srepositories/%s/tags", host, repository)
```

获取指定 repository 中的所有 tag 信息之后，Docker Daemon 根据 tag 对应 layer 的 ID，找到 ImgData，并填充 ImgData 中的 Tag 属性。此时，RepositoryData 的 ImgList 属性中，有的 ImgData 对象中有 Tag 内容，有的 ImgData 对象中没有 Tag 内容。这也和实际情况相符，如果下载一个 ubuntu:14.04 镜像，该镜像的 rootfs 中只有最上层的 layer 才有 tag 信息，这一个 layer 的父镜像并不一定存在 tag 信息。

3. pullImage

Docker Daemon 下载 Docker 镜像是通过镜像 ID 来完成的。GetRepositoryData 和 GetRemoteTags 则成功完成了用户传入的 repository 和 tag 信息与镜像 ID 之间的转换。如请求 docker pull ubuntu:14.04 中，repository 为 ubuntu，tag 为 14.04，则对应的镜像 ID 为 2d24f826。需要注意的是，ubuntu:14.04 镜像的 layer 并不是一成不变的，Docker Hub 上关于该镜像的内容仍然会更新。

Docker Daemon 获得下载镜像的镜像 ID 之后，首先查验 pullingPool，判断是否有其他 Docker Client 同样发起了该镜像的下载请求，若没有 Docker Daemon 才继续下载任务。

执行 pullImage 函数的源码实现位于 ./docker/graph/pull.go#L159，如下：

```
s.pullImage(r, out, img.ID, ep, repoData.Tokens, sf)
```

而 pullImage 函数的定义位于 ./docker/graph/pull.go#L214-L301。图 9-3 中，我们可以看到 pullImage 函数的执行可以分为 4 个步骤：GetRemoteHistory、GetRemoteImageJSON、GetRemoteImageLayer 与 s.graph.Register()。

GetRemoteHistory 的作用很好理解，既然 Docker Daemon 已经通过 GetRepositoryData 和 GetRemoteTags 找出了指定 tag 的 image id，那么 Docker Daemon 所需完成的工作为下载该 image 及其所有的祖先 image。GetRemoteHistory 负责获取指定 image 及其所有祖先 image 的 id。

GetRemoteHistory 的源码实现位于 ./docker/registry/session.go#L72-L101。

获取所有的 image id 之后，对于每一个 image id，Docker Daemon 都开始下载该 image 的全部内容。一个 layer 的 Docker Image 包括两个方面的内容：image json 信息以及 image layer 信息。Docker 所有 image 的 json 信息都由函数 GetRemoteImageJSON 来获取。分析 GetRemoteImageJSON 之前，有必要阐述清楚什么是 Docker Image 的 json 信息。

Docker Image 的 json 信息是一个非常重要的概念。这部分 json 唯一标志一个 image，不仅标志 image 的 id，还标志 image 所在 layer 对应的 config 配置信息。为了理解以上内容，可以举一个例子：docker build。命令 docker build 用于通过指定的 Dockerfile 来创建一个 Docker 镜像；对于 Dockerfile 中所有的命令，Docker Daemon 都会为其创建一个新的 image，如：RUN apt-get update, ENV path=/bin, WORKDIR /home 等。对于命令 RUN apt-get update，Docker Daemon 需要执行 apt-get update 操作，对应的 rootfs 上必定会有内容更新，导致新建的 image 所代表的 layer 中有新添加的内容。而如 ENV path=/bin, WORKDIR /home 这样的命令，仅仅配置了一些容器的运行参数，并没有镜像内容的更新，对于这种情况，Docker Daemon 同样创建一个新的 layer，并且这个新的 layer 中内容为空，而命令内容会在这层 image 的 json 信息中做更新。总之，可以认为 Docker 的 image 包含两部分内容：image 的 json 信息、layer 内容。当 layer 内容为空时，image 的 json 信息被更新。第 11 章将重点分析 Docker 中 dockerbuild 的实现。

清楚 Docker image 的 json 信息之后，理解 GetRemoteImageJSON 函数的作用就变得十分容易。执行 GetRemoteImageJSON 的源码实现位于 ./docker/graph/pull.go#L243，如下：

```
imgJSON, imgSize, err = r.GetRemoteImageJSON(id, endpoint, token)
```

GetRemoteImageJSON 返回的对象 imgJSON 代表 image 的 json 信息，imgSize 代表镜像的大小。通过 imgJSON 对象，Docker Daemon 立即创建一个 image 对象，创建 image 对象的源码实现位于 ./docker/graph/pull.go#L251，如下：

```
img, err = image.NewImgJSON(imgJSON)
```

而 NewImgJSON 函数位于包 image 中，函数返回类型为一个 Image 对象，Image 类型的定义如下：

```
type Image struct {
    ID              string              `json:"id"`
    Parent          string              `json:"parent,omitempty"`
    Comment         string              `json:"comment,omitempty"`
    Created         time.Time           `json:"created"`
    Container       string              `json:"container,omitempty"`
    ContainerConfig runconfig.Config    `json:"container_config,omitempty"`
    DockerVersion   string              `json:"docker_version,omitempty"`
    Author          string              `json:"author,omitempty"`
    Config          *runconfig.Config   `json:"config,omitempty"`
    Architecture    string              `json:"architecture,omitempty"`
    OS              string              `json:"os,omitempty"`
    Size            int64

    graph Graph
}
```

返回 img 对象，则说明关于该 image 的所有元数据已经保存完毕，由于还缺少 image 的 layer 中包含的内容，因此下一个步骤即为下载镜像 layer 的内容，调用的函数为 GetRemoteImageLayer，函数的源码位于 ./docker/graph/pull.go#L270，如下：

```
layer, err := r.GetRemoteImageLayer(img.ID, endpoint, token, int64(imgSize))
```

GetRemoteImageLayer 函数返回当前 image 的 layer 内容。image 的 layer 内容指的是：该 image 在 parent image 之上做的文件系统内容更新，包括文件的增添、删除、修改等。至此，image 的 json 信息以及 layer 内容均被 Docker Daemon 获取，这意味着一个完整的 image 已经下载完毕。下载 image 完毕之后，并不意味着 Docker Daemon 关于 Docker 镜像下载的 Job 就此结束，Docker Daemon 仍然需要对下载的 image 进行存储管理，以便 Docker Daemon 在执行其他（如创建容器等）Job 时，能够方便地使用这些 image。

Docker Daemon 在 graph 中注册 image 的源码实现位于 ./docker/graph/pull.go#L283-L285，如下：

```
err = s.graph.Register(imgJSON,utils.ProgressReader(layer, imgSize, out, sf,
false, utils.TruncateID(id), "Downloading"),img)
```

Docker Daemon 通过 graph 存储 image 是一个很重要的环节。Docker 在 1.2.0 版本中可以通过 aufs、DevMapper 以及 BTRFS 来进行 image 的存储。在 Linux 3.18-rc2 版本中，OverlayFS 已经被合入内容主线，故从 Docker 1.4.0 版本开始，Docker 的 image 支持 OverlayFS 的存储方式。

Docker 镜像的存储在 Docker 中是较为独立且重要的内容，第 10 章将对 Docker 镜像的存储进行深入分析。

4. 配置 TagStore

Docker 镜像下载完毕之后，Docker Daemon 需要在 TagStore 中指定的 repository 中添加

相应的 tag。每当用户查看本地镜像时，都可以从 TagStore 的 repository 中查看所有含 tag 信息的 image。

Docker Daemon 配置 TagStore 的源码实现位于 ./docker/graph/pull.go#L206，如下：

```
if err := s.Set(localName, tag, id, true); err != nil {
    return err
}
```

TagStore 类型的 Set 函数定义位于 ./docker/graph/tags.go#L174-L205。Set 函数的执行流程与简要介绍如图 9-5 所示。

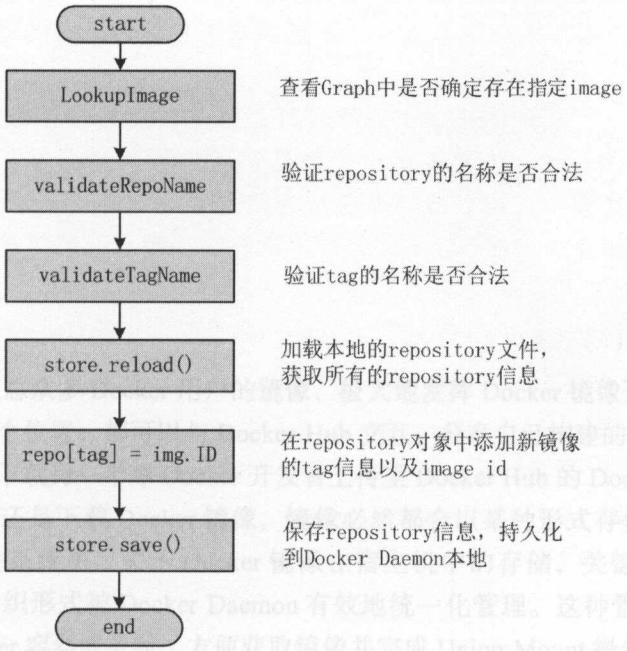


图 9-5　TagStore 中 Set 函数的执行流程及简介

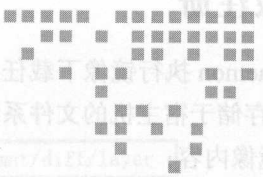
当 Docker Daemon 将已下载的 Docker 镜像信息同步到 repository 之后，Docker 下载镜像的 Job 就全部完成，Docker Daemon 返回响应至 Docker Server，Docker Server 返回响应至 Docker Client。本地的 repository 文件位于 Docker 的根目录，根目录一般为 /var/lib/docker，如果使用 aufs 的 graphdriver，则 repository 文件名为 repositories-aufs。

9.6　总结

Docker 镜像给 Docker 容器的运行带来了无限的可能性，诸如 Docker Hub 之类的 Docker Registry 又使得 Docker 镜像在全球的开发者之间共享。Docker 镜像的下载是使用 Docker 的

第一个步骤。Docker 爱好者若能熟练掌握其中的原理，必定能对 Docker 的很多概念有更为清晰的认识，对 Docker 容器的运行、管理等均是有百利而无一害。

Docker 镜像的下载需要 Docker Client、Docker Server、Docker Daemon 以及 Docker Registry 四者协同合作完成。本章从源码的角度分析了四者各自扮演的角色，分析过程中还涉及多种 Docker 概念，如 repository、tag、TagStore、session、image、layer、image json 和 graph 等。



Chapter 10

第 10 章 Docker 镜像存储

10.1 引言

Docker Hub 汇总众多 Docker 用户的镜像，极大地发挥 Docker 镜像开放的思想。Docker 用户在全球任意一个位置，都可以与 Docker Hub 交互，分享自己构建的镜像至 Docker Hub，当然，也完全可以下载另一半球 Docker 开发者上传至 Docker Hub 的 Docker 镜像。

无论是上传，还是下载 Docker 镜像，镜像必然都会以某种形式存储在 Docker Daemon 所在的宿主机文件系统中。关于 Docker 镜像在宿主机中的存储，关键点在于：在本地文件系统中以何种组织形式被 Docker Daemon 有效地统一化管理。这种管理可以使得 Docker Daemon 创建 Docker 容器服务时，方便获取镜像并完成 Union Mount 操作，为容器准备初始化的文件系统。

本章主要从 Docker 1.2.0 源码的角度，分析 Docker Daemon 下载镜像过程中存储 Docker 镜像的环节。本章分析的内容有以下 5 部分：

1）概述 Docker 镜像存储的执行入口，并简要介绍存储流程的四个步骤；

2）验证镜像 ID 的有效性；

3）创建镜像存储路径；

4）存储镜像内容；

5）在 Graph 中注册镜像 ID。

10.2 镜像注册

Docker Daemon 执行镜像下载任务时，从 Docker Registry 处下载指定镜像之后，仍需要将镜像合理地存储于宿主机的文件系统中。更为具体而言，存储工作分为两个部分：

1）存储镜像内容；

2）在 Graph 中注册镜像信息。

提到镜像内容，需要强调的是，每一个 layer 的 DockerImage 内容都可以认为由两个部分组成：镜像中每一个 layer 中存储的文件系统内容，这部分内容一般可以认为是未来 Docker 容器的静态文件内容；另一部分内容指的是容器的 json 文件，json 文件代表的信息除了容器的基本属性信息之外，还包括未来容器运行时的动态信息，如 ENV 等信息。

存储镜像内容，意味着 Docker Daemon 所在宿主机上已经存在镜像的所有内容，除此之外，Docker Daemon 仍需要对所存储的镜像进行统计备案，以便用户在后续的镜像管理与使用过程中，可以有据可循。为此，Docker Daemon 设计了 Graph，使用 Graph 来接管这部分的工作。Graph 负责记录有哪些镜像已经正确存储，供 Docker Daemon 调用。

Docker Daemon 执行 CmdPull 任务的 pullImage 阶段时，实现 Docker 镜像存储与记录的源码位于 ./docker/graph/pull.go#L283-L285，如下：

```
err = s.graph.Register(imgJSON,utils.ProgressReader(layer, imgSize, out, sf,
false, utils.TruncateID(id), "Downloading"),img)
```

以上源码的实现，实际调用了函数 Register，Register 函数的定义位于 ./docker/graph/graph.go#L162-L218，如下：

```
func (graph *Graph) Register(jsonData []byte, layerData archive.ArchiveReader,
img *image.Image) (err error)
```

分析以上 Register 函数的定义，可以得出以下内容：

1）函数名称为 Register；

2）函数调用者类型为 Graph；

3）传入函数的参数有 3 个：第一个为 jsonData，类型为数组；第二个为 layerData，类型为 archive.ArchiveReader；第三个为 img，类型为 *image.Image；

4）函数返回对象为 err，类型为 error。

Register 函数的运行流程如图 10-1 所示。

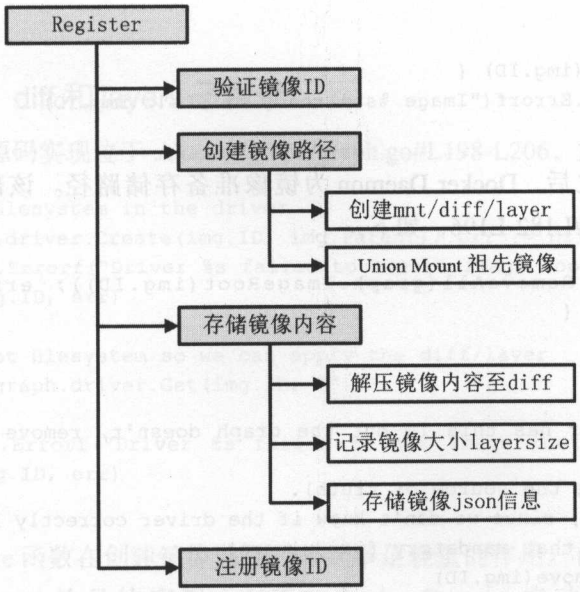


图 10-1　Register 函数执行流程

10.3　验证镜像 ID

Docker 镜像注册的第一个步骤是验证 Docker 镜像的 ID。此步骤确保镜像 ID 命名的合法性。就功能而言，这部分内容提高了 Docker 镜像存储环节的鲁棒性。验证镜像 ID 由三个环节组成。

1）验证镜像 ID 的合法性；

2）验证镜像是否已存在；

3）初始化镜像目录。

验证镜像 ID 的合法性使用包 utils 中的 ValidateID 函数完成，实现源码位于 ./docker/graph/graph.go#L171-L173，如下：

```
if err := utils.ValidateID(img.ID); err != nil {
    return err
}
```

ValidateID 函数的实现过程中，Docker Dameon 检验了镜像 ID 是否为空，以及镜像 ID 中是否存在字符“:”，以上两种情况只要其中之一成立，Docker Daemon 就认为镜像 ID 不合法，不予执行后续内容。

镜像 ID 的合法性验证完毕之后，Docker Daemon 接着验证镜像是否已经存在于 Graph 中。若该镜像已经存在于 Graph 中，则 Docker Daemon 返回相应错误，不予执行后续内容。代码

实现如下：

```
if graph.Exists(img.ID) {
    return fmt.Errorf("Image %s already exists", img.ID)
}
```

验证工作完成之后，Docker Daemon 为镜像准备存储路径。该部分源码实现位于 ./docker/graph/graph.go#L182-L196，如下：

```
if err := os.RemoveAll(graph.ImageRoot(img.ID)); err != nil && !os.
IsNotExist(err) {
    return err
}

// If the driver has this ID but the graph doesn't, remove it from the driver
to start fresh.
// (the graph is the source of truth).
// Ignore errors, since we don't know if the driver correctly returns ErrNotExist.
// (FIXME: make that mandatory for drivers).
graph.driver.Remove(img.ID)

tmp, err := graph.Mktemp("")
defer os.RemoveAll(tmp)
if err != nil {
    return fmt.Errorf("Mktemp failed: %s", err)
}
```

Docker Daemon 为镜像初始化存储路径，实则首先删除属于新镜像的存储路径，即如果该镜像路径已经在文件系统中存在，立即删除该路径，确保存储镜像时不会出现路径冲突问题；接着还删除 graph.driver 中的指定内容，即如果该镜像在 graph.driver 中存在，卸载该镜像在宿主机上的目录，并将该目录完全删除。以 AUFS 这种类型的 graphdriver 为例，镜像内容存放在 /var/lib/docker/aufs/diff 目录下，而镜像会被挂载至目录 /var/lib/docker/aufs/mnt 下的指定位置。

至此，验证 Docker 镜像 ID 的工作已经完成，并且 Docker Daemon 已经完成对镜像存储路径的初始化，使得后续 Docker 镜像存储时存储路径不会冲突，graph.driver 对该镜像的挂载也不会冲突。

10.4 创建镜像路径

创建镜像路径，是镜像存储流程中的一个必备环节，这一环节直接让 Docker 使用者了解以下概念：镜像以何种形式存在于本地文件系统的何处。创建镜像路径完毕之后，Docker Daemon 首先将镜像的所有祖先镜像通过 aufs 文件系统挂载至 mnt 下的指定点，最终直接返回镜像所在 rootfs 的路径，以便后续直接在该路径下解压 Docker 镜像的具体内容（只包含

layer 内容)。

10.4.1　创建 mnt、diff 和 layers 子目录

创建镜像路径的源码实现位于 ./docker/graph/graph.go#L198-L206，如下：

```
// Create root filesystem in the driver
if err := graph.driver.Create(img.ID, img.Parent); err != nil {
    return fmt.Errorf("Driver %s failed to create image rootfs %s: %s", graph.
    driver, img.ID, err)
}
// Mount the root filesystem so we can apply the diff/layer
rootfs, err := graph.driver.Get(img.ID, "")
if err != nil {
    return fmt.Errorf("Driver %s failed to get image rootfs %s: %s", graph.
    driver, img.ID, err)
}
```

以上源码中 Create 函数在创建镜像路径时起到举足轻重的作用。首先分析 graph.driver.Create(img.ID, img.Parent) 的具体实现。由于在 Docker Daemon 启动时，注册了具体的 graphdriver，故 graph.driver 实际的值为具体注册的 driver。方便起见，本章内容全部以 aufs 类型为例，即在 graph.driver 为 aufs 的情况下，阐述 Docker 镜像的存储。在 Ubuntu 14.04 系统上，Docker Daemon 的根目录一般为 /var/lib/docker, 而 aufs 类型 driver 的镜像存储路径一般为 /var/lib/docker/aufs。

aufs 这种联合文件系统的实现，在合并多个镜像时起到至关重要的作用。首先介绍 Docker Daemon 如何为镜像创建镜像路径，以便支持通过 aufs 来 union 镜像。aufs 模式下，graph.driver.Create(img.ID, img.Parent) 的具体源码实现位于 ./docker/daemon/graphdriver/aufs/aufs.go#L161-L190，如下：

```
// Three folders are created for each id
// mnt, layers, and diff
func (a *Driver) Create(id, parent string) error {
    if err := a.createDirsFor(id); err != nil {
        return err
    }
    // Write the layers metadata
    f, err := os.Create(path.Join(a.rootPath(), "layers", id))
    if err != nil {
        return err
    }
    defer f.Close()

    if parent != "" {
        ids, err := getParentIds(a.rootPath(), parent)
        if err != nil {
            return err
```

```
            }

            if _, err := fmt.Fprintln(f, parent); err != nil {
                return err
            }
            for _, i := range ids {
                if _, err := fmt.Fprintln(f, i); err != nil {
                    return err
                }
            }
        }
        return nil
    }
```

在 Create 函数的实现过程中，createDirsFor 函数在 Docker Daemon 根目录下的 aufs 目录 /var/lib/docker/aufs 中，创建指定的镜像目录。若当前 aufs 目录下还不存在 mnt、diff 这两个目录，则会首先创建 mnt、diff 这两个目录，并在这两个目录下分别创建代表镜像内容的文件夹，文件夹名为镜像 ID，文件权限为 0755。假设下载镜像的镜像 ID 为 image_ID，则创建完毕之后，文件系统中的文件为 /var/lib/docker/aufs/mnt/image_ID 与 /var/lib/docker/aufs/diff/image_ID。回到 Create 函数中，执行完 createDirsFor 函数之后，随即在 aufs 目录下创建了 layers 目录，并在 layers 目录下创建 image_ID 文件。

如此一来，在 aufs 下的三个子目录 mnt、diff 以及 layers 中，分别创建了名为镜像名 image_ID 的文件。继续深入分析之前，我们直接来看 Docker 对这三个目录 mnt、diff 以及 layers 的描述，如图 10-2 所示。

```
aufs driver directory structure
.
├── layers // Metadata of layers
│   ├── 1
│   ├── 2
│   └── 3
├── diff  // Content of the layer
│   ├── 1  // Contains layers that need to be mounted for the id
│   ├── 2
│   └── 3
└── mnt    // Mount points for the rw layers to be mounted
    ├── 1
    ├── 2
    └── 3
```

图 10-2　aufs driver 目录结构图

简要分析图 10-2，其中，layers、diff 以及 mnt 为目录 /var/lib/docker/aufs 下的三个子目录，1、2、3 是镜像 ID，分别代表三个镜像，三个目录下的 1 均代表同一个镜像 ID。其中 layers 目录下保留每一个镜像的元数据，这些元数据是这个镜像的祖先镜像 ID 列表；diff 目录下存储每一个镜像所在的 layer，具体包含的文件系统内容；mnt 目录下每一个文件都是一个镜像 ID，代表在该层镜像之上挂载的可读写 layer。因此，下载的镜像中与文件系统相关的具体内容，都会存储在 diff 目录下的某个镜像 ID 目录下。

再次回到 Create 函数，此时 mnt、diff 以及 layers 三个目录下的镜像 ID 文件已经创建完毕。下一步需要完成的是：为 layers 目录下的镜像 ID 文件填充元数据。元数据内容为该镜像的所有祖先镜像 ID 列表。填充元数据的流程如下：

1）Docker Daemon 首先通过 `f, err := os.Create(path.Join(a.rootPath(), "layers", id))` 打开 layers 目录下镜像 ID 文件；

2）然后，通过 `ids, err := getParentIds(a.rootPath(), parent)` 获取父镜像的祖先镜像 ID 列表 ids；

3）其次，将父镜像 ID 写入文件 f；

4）最后，将父镜像的祖先镜像 ID 列表 ids 写入文件 f。

最终的结果是：该镜像的所有祖先镜像的镜像 ID 信息都写入 layers 目录下该镜像 ID 文件中。

10.4.2　挂载祖先镜像并返回根目录

Create 函数执行完毕，意味着创建镜像路径并配置镜像元数据完毕，接着 Docker Daemon 返回镜像的根目录，源码实现如下：

```
rootfs, err := graph.driver.Get(img.ID, "")
```

Get 函数看似返回了镜像的根目录 rootfs，实则执行了更为重要的内容——挂载祖先镜像文件系统。具体而言，Docker Daemon 为当前层的镜像完成所有祖先镜像的 Union Mount。挂载完毕之后，当前镜像的 read-write 层位于 /var/lib/docker/aufs/mnt/image_ID。Get 函数的具体实现位于 ./docker/daemon/graphdriver/aufs/aufs.go#L247-L278，如下：

```
func (a *Driver) Get(id, mountLabel string) (string, error) {
    ids, err := getParentIds(a.rootPath(), id)
    if err != nil {
        if !os.IsNotExist(err) {
            return "", err
        }
        ids = []string{}
    }

    // Protect the a.active from concurrent access
    a.Lock()
    defer a.Unlock()

    count := a.active[id]

    // If a dir does not have a parent ( no layers )do not try to mount
    // just return the diff path to the data
    out := path.Join(a.rootPath(), "diff", id)
    if len(ids) > 0 {
        out = path.Join(a.rootPath(), "mnt", id)
```

```
            if count == 0 {
                if err := a.mount(id, mountLabel); err != nil {
                    return "", err
                }
            }
        }

        a.active[id] = count + 1

        return out, nil
    }
```

分析以上 Get 函数的定义，可以得出以下内容：

1）函数名为 Get；

2）函数调用者类型为 Driver；

3）传入函数的参数有两个：id 与 mountlabel；

4）函数返回内容有两部分：string 类型的镜像根目录与错误对象 error。

清楚 Get 函数的定义，再来看 Get 函数的实现。分析 Get 函数实现时，有三个部分较为关键，分别是 Driver 实例 a 的 active 属性、mount 操作以及返回值 out。

首先分析 Driver 实例 a 的 active 属性。分析 active 属性之前，需要追溯到 aufs 类型的 graphdriver 中 Driver 类型的定义以及 graphdriver 与 Graph 的关系。两者的关系如图 10-3 所示。

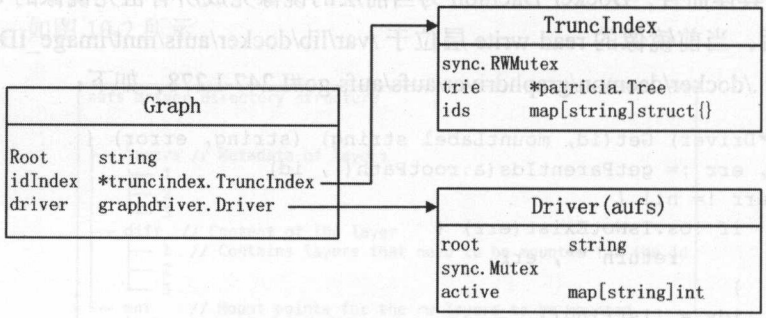


图 10-3 Graph 与 graphdriver 的关系

Driver 类型的定义位于 ./docker/daemon/graphdriver/aufs/aufs#L53-L57，如下：

```
type Driver struct {
    root       string
    sync.Mutex // Protects concurrent modification to active
    active     map[string]int
}
```

Driver 结构体中 root 属性代表 graphdriver 所在的根目录，即 /var/lib/docker/aufs。active 属性为 map 类型，key 为 string，具体运用时 key 为 DockerImage 的 ID，value 为 int 类型，

代表该层镜像 layer 被引用的次数总和。Docker 镜像技术中，某一个 layer 的 Docker 镜像被引用一次，则 active 属性中 key 为该镜像 ID 的 value 值会累加 1。用户执行镜像删除操作时，Docker Dameon 会检查该 Docker 镜像的引用次数是否为 0，若引用次数为 0，则可以彻底删除该镜像，若非零，则仅仅将 active 属性中引用参数减 1。属性 sync.Mutex 用于多个 Job 同时操作 active 属性时，确保 active 数据的同步工作。

接着，进入挂载操作的分析。一旦 Get 参数传入的镜像 ID 参数不是一个基础镜像，那么说明该镜像存在父镜像，Docker Daemon 需要将该镜像所有的祖先镜像都挂载到指定的位置，指定位置为 /var/lib/docker/aufs/mnt/image_ID。所有祖先镜像的原生态文件系统内容分别位于 /var/lib/docker/aufs/diff/<ID>。其中 mount 函数用于实现该部分描述的功能，挂载的过程包含很多与 aufs 文件系统相关的参数配置与系统调用。

最后，Get 函数返回 out 与 nil。其中 out 的值为 /var/lib/docker/aufs/mnt/image_ID，即使用该层 Docker 镜像时其根目录所在路径，也可以认为是镜像的 RW 层所在路径，但一旦该层镜像之上还有镜像，那么在挂载后者之后，在上层镜像看来，下层镜像仍然是只读文件系统。

10.5　存储镜像内容

存储镜像内容，意味着 Docker Daemon 以及验证过镜像 ID，同时还为镜像准备了存储路径，并返回了其所有祖先镜像执行 Union Mount 操作后的路径。万事俱备，只欠“镜像内容的存储”。

Docker Daemon 为了存储镜像具体内容完成的工作很简单，仅仅是通过某种合适的方式将两部分内容存储于本地文件系统并进行有效管理，两部分内容是镜像压缩内容、镜像 json 信息。

存储镜像内容的源码实现位于 ./docker/graph/graph.go#L209-L211，如下：

```
if err := image.StoreImage(img, jsonData, layerData, tmp, rootfs); err != nil {
    return err
}
```

其中，StoreImage 函数的定义位于 ./docker/docker/image/image.go#L74，如下：

```
func StoreImage(img *Image, jsonData []byte, layerData archive.ArchiveReader,
root, layer string) error {
```

分析 StoreImage 函数的定义，可以得出以下信息：

1）函数名称：StoreImage；

2）传入函数的参数名：img、jsonData、layerData、root、layer；

3）函数返回类型 error。

传入参数的含义如表 10-1 所示。

表 10-1 StoreImage 函数的传入参数

参数名称	参数含义
img	通过下载的 imgJSON 信息创建出的 Image 对象实例
jsonData	Docker Daemon 之前下载的 imgJSON 信息
layerData	镜像作为一个 layer 的压缩包，包含镜像的具体文件内容
root	graphdriver 根目录下创建的临时文件“_tmp”，值为 /var/lib/docker/aufs/_tmp
layer	挂载完所有祖先镜像之后，该镜像在 mnt 目录下的路径

掌握 StoreImage 函数传入参数的含义之后，理解其实现就十分简单。总体而言，StoreImage 亦可以分为三个步骤：

1）解压镜像内容 layerData 至 diff 目录；

2）收集镜像所占空间大小，并记录；

3）将 jsonData 信息写入临时文件。

下面详细介绍三个步骤的实现。

10.5.1 解压镜像内容

StoreImage 函数传入的镜像内容是一个压缩包，Docker Daemon 理应在镜像存储时将其解压，为后续创建容器时直接使用镜像创造便利。

既然是解压镜像内容，那么这项任务的完成，除了需要代表镜像的压缩包之后，还需要解压任务的目标路径，以及解压时的参数。压缩包为传入 StoreImage 的参数 layerData，而目标路径为 /var/lib/docker/aufs/diff/<image_ID>。解压流程的源代码位于 ./docker/docker/image/image.go#L85-L120，如下：

```
// If layerData is not nil, unpack it into the new layer
if layerData != nil {
    if differ, ok := driver.(graphdriver.Differ); ok {
        if err := differ.ApplyDiff(img.ID, layerData); err != nil {
            return err
        }

        if size, err = differ.DiffSize(img.ID); err != nil {
            return err
        }
    } else {
        start := time.Now().UTC()
        log.Debugf("Start untar layer")
        if err := archive.ApplyLayer(layer, layerData); err != nil {
            return err
        }
        log.Debugf("Untar time: %vs", time.Now().UTC().Sub(start).Seconds())
```

```
        if img.Parent == "" {
            if size, err = utils.TreeSize(layer); err != nil {
                return err
            }
        } else {
            parent, err := driver.Get(img.Parent, "")
            if err != nil {
                return err
            }
            defer driver.Put(img.Parent)
            changes, err := archive.ChangesDirs(layer, parent)
            if err != nil {
                return err
            }
            size = archive.ChangesSize(layer, changes)
        }
    }
}
```

可见，当镜像内容 layerData 不为空时，Docker Daemon 需要为镜像压缩包执行解压工作。以 aufs 这种 graphdriver 为例，一旦 aufs driver 实现了 graphdriver 包中的接口 Diff，Docker Daemon 就会使用 aufs driver 的接口方法实现后续的解压操作。解压操作的源代码如下：

```
if differ, ok := driver.(graphdriver.Differ); ok {
    if err := differ.ApplyDiff(img.ID, layerData); err != nil {
        return err
    }

    if size, err = differ.DiffSize(img.ID); err != nil {
        return err
    }
}
```

以上代码实现了镜像压缩包的解压与镜像所占空间大小的统计。代码 differ.ApplyDiff(img.ID, layerData) 将 layerData 解压至目标路径。理清目标路径，且看 aufs driver 中 ApplyDiff 的实现，位于 ./docker/docker/daemon/graphdriver/aufs/aufs.go#L304-L306，如下：

```
func (a *Driver) ApplyDiff(id string, diff archive.ArchiveReader) error {
    return archive.Untar(diff, path.Join(a.rootPath(), "diff", id), nil)
}
```

解压过程中，Docker Daemon 通过 aufs driver 的根目录 /var/lib/docker/aufs、diff 目录与镜像 ID，拼接出镜像的解压路径，并执行解压任务。举例说明 diff 文件的作用，镜像 27d474 解压后的内容如图 10-4 所示。

回到 StoreImage 函数的执行流中，ApplyDiff 任务完成之后，Docker Daemon 通过 DiffSize 开启镜像磁盘空间统计任务。

```
root@vm:/var/lib/docker/aufs/diff/27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3# ll
total 100
drwxr-xr-x  21 root root  4096 Feb  1 16:11 ./
drwxr-xr-x 165 root root 20480 Mar 29 11:12 ../
drwxr-xr-x   2 root root  4096 Jan 29 00:28 bin/
drwxr-xr-x   2 root root  4096 Apr 11  2014 boot/
drwxr-xr-x   3 root root  4096 Jan 29 00:28 dev/
drwxr-xr-x  61 root root  4096 Jan 29 00:28 etc/
drwxr-xr-x   2 root root  4096 Apr 11  2014 home/
drwxr-xr-x  12 root root  4096 Jan 29 00:28 lib/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 lib64/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 media/
drwxr-xr-x   2 root root  4096 Apr 11  2014 mnt/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 opt/
drwxr-xr-x   2 root root  4096 Apr 11  2014 proc/
drwx------   2 root root  4096 Jan 29 00:28 root/
drwxr-xr-x   7 root root  4096 Jan 29 00:28 run/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 sbin/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 srv/
drwxr-xr-x   2 root root  4096 Mar 13  2014 sys/
drwxrwxrwt   2 root root  4096 Jan 29 00:29 tmp/
drwxr-xr-x  10 root root  4096 Jan 29 00:28 usr/
drwxr-xr-x  11 root root  4096 Jan 29 00:28 var/
root@vm:/var/lib/docker/aufs/diff/27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3#
```

图 10-4 镜像解压后示意图

10.5.2 收集镜像大小并记录

Docker Daemon 接管镜像存储之后，Docker 镜像被解压到指定路径并非意味着“任务完成”。Docker Daemon 还统计了镜像所占磁盘空间的大小，以便记录镜像信息，最终将这类信息传递给 Docker 用户。

镜像所占磁盘空间大小的统计与记录，实现过程简单且有效，源代码位于 ./docker/docker/image/image.go#L122-L125，如下：

```
img.Size = size
if err := img.SaveSize(root); err != nil {
    return err
}
```

首先 Docker Daemon 将镜像大小收集起来，更新 Image 类型实例 img 的 Size 属性，然后通过 img.SaveSize(root) 将镜像大小写入 root 目录，由于传入的 root 参数为临时目录 _tmp，即写入临时目录 _tmp 下。实现 SaveSize 函数的源码如下：

```
func (img *Image) SaveSize(root string) error {
    if err := ioutil.WriteFile(path.Join(root, "layersize"), []byte(strconv.
    Itoa(int(img.Size))), 0600); err != nil {
        return fmt.Errorf("Error storing image size in %s/layersize: %s",
        root, err)
    }
    return nil
}
```

SaveSize 函数在 root 目录（临时目录 /var/lib/docker/graph/_tmp）下创建文件 layersize，

并写入镜像大小的值 img.Size。

10.5.3　存储 jsonData 信息

在 Docker 镜像中，jsonData 是一个非常重要的概念。在笔者看来，Docker 的镜像并非只是 Docker 容器文件系统中的文件内容，同时还包括 Docker 容器运行的动态信息。这里的动态信息更多地是为了适配 Dockerfile 的标准。以 Dockerfile 中的 ENV 参数为例，ENV 指定了 Docker 容器运行时内部进程的环境变量。而这些只有容器运行时才存在的动态信息，并不会被记录在静态的镜像文件系统中，而是以 jsonData 的形式先存储在宿主机的文件系统中，并与镜像文件系统泾渭分明，存储在不同的位置。当 Docker Daemon 启动 Docker 容器时，Docker Daemon 会准备好挂载完毕的镜像文件系统环境；接着加载 jsonData 信息，并在运行 Docker 容器内部进程时，使用动态的 jsonData 内部信息为容器内部进程配置环境。

当 Docker Daemon 下载 Docker 镜像时，关于每一个镜像的 jsonData 信息均会被下载至宿主机。通过以上 jsonData 的功能描述可以发现，这部分信息的存储同样扮演重要的角色。关于 Docker Daemon 如何存储 jsonData 信息，实现源码位于 ./docker/docker/image/image.go#L128-L139，如下：

```
if jsonData != nil {
    if err := ioutil.WriteFile(jsonPath(root), jsonData, 0600); err != nil {
        return err
    }
} else {
    if jsonData, err = json.Marshal(img); err != nil {
        return err
    }
    if err := ioutil.WriteFile(jsonPath(root), jsonData, 0600); err != nil {
        return err
    }
}
```

可见 Docker Daemon 将 jsonData 写入了文件 jsonPath(root) 中，并且把该文件的权限设置为 0600。而 jsonPath(root) 的实现如下，即在 root 目录（/var/lib/docker/graph/_tmp 目录）下创建文件 json：

```
func jsonPath(root string) string {
    return path.Join(root, "json")
}
```

镜像大小信息 layersize 信息统计完毕，jsonData 信息也成功记录，两者的存储文件均位于 /var/lib/docker/graph/_tmp 下，文件名分别为 layersize 和 json。使用临时文件夹来存储这部分信息并非偶然，10.6 节将阐述其中的原因。

10.6 注册镜像 ID

Docker Daemon 执行完镜像的 StoreImage 操作，回到 Register 函数之后，执行镜像的 commit 操作，即完成镜像在 Graph 中的注册。

注册镜像的代码实现位于 ./docker/docker/graph/graph.go#L212-L216，如下：

```
// commit
if err := os.Rename(tmp, graph.ImageRoot(img.ID)); err != nil {
      return err
}
graph.idIndex.Add(img.ID)
```

10.5 节存储镜像过程中使用到的临时文件 _tmp 在注册镜像环节有所体现。关于镜像的注册行为，第一步就是将 _tmp 文件（/var/lib/docker/graph/_tmp）重命名为 graph.ImageRoot(img.ID)，实则为 /var/lib/docker/graph/<img.ID>。使得 Docker Daemon 在而后的操作中可以通过 img.ID 在 /var/lib/docker/graph 目录下搜索到相应镜像的 json 文件与 layersize 文件。

成功为 json 文件与 layersize 文件配置完正确的路径之后，Docker Daemon 执行的最后一个步骤为：添加镜像 ID 至 graph.idIndex。源代码实现是 graph.idIndex.Add(img.ID)，Graph 中 idIndex 类型为 *truncindex.TruncIndex，TruncIndex 的定义位于 ./docker/docker/pkg/truncindex/truncindex.go#L22-L28，如下：

```
// TruncIndex allows the retrieval of string identifiers by any of their unique prefixes.
// This is used to retrieve image and container IDs by more convenient shorthand prefixes.
type TruncIndex struct {
     sync.RWMutex
     trie *patricia.Trie
     ids  map[string]struct{}
}
```

Docker 用户使用 Docker 镜像时，一般可以通过指定镜像 ID 来定位镜像，如 Docker 官方 的 mongo:2.6.1 镜 像 ID 为 c35c0961174d51035d6e374ed9815398b779296b5f0ffceb7613c8199383f4b1，该 ID 长度为 64。当 Docker 用户指定运行这个 Mongo 镜像 Repository 中 tag 为 2.6.1 的镜像时，完全可以通过 64 位的镜像 ID 来指定，如下：

```
docker run -it c35c0961174d51035d6e374ed9815398b779296b5f0ffceb7613c8199383f4b1
/bin/bash
```

然而，记录如此长的镜像 ID，对于 Docker 用户来说，稍显不切实际，而 TruncIndex 的概念则极大地帮助 Docker 用户可以通过简短的 ID 定位到指定的镜像，使得 Docker 镜像的使用变得尤为方便。原理是：Docker 用户指定镜像 ID 的前缀，只要前缀满足在全局所有的镜像 ID 中唯一，Docker Daemon 就可以通过 TruncIndex 定位到唯一的镜像 ID。而 graph.idIndex.Add(img.ID) 正式完成将 img.ID 添加并保存至 TruncIndex 中。

为了达到上一条命令的效果，Docker 用户完全可以使用 TruncIndex 的方式，当然，前提是 c35 这个字符串作为前缀全局唯一，命令如下：

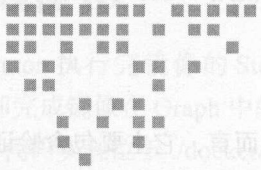
```
docker run -it c35 /bin/bash
```

至此，Docker 镜像存储的整个流程已经完成。概括而言，它主要包含验证镜像、存储镜像、注册镜像三个步骤。

10.7　总结

Docker 镜像的存储，使得 Docker Hub 上的镜像能够遍布世界各地变为现实。Docker 镜像在 Docker Registry 中的存储方式与本地化的存储方式并非一致。Docker Daemon 必须针对自身的 graphdriver 类型，选择适配的存储方式，实施镜像的存储。本章也在不断强调一个事实，即 Docker 镜像并非仅仅包含文件系统中的静态文件，除此之外，它还包含镜像的 json 信息，json 信息中有 Docker 容器的配置信息，如暴露端口、环境变量等。

可以说 Docker 容器的运行严重依赖于 Docker 镜像，因此了解 Docker 镜像的由来就变得尤为重要。Docker 镜像的下载、Docker 镜像的打包以及构建新的 Docker 镜像，都无法跳出镜像存储的范畴。Docker 镜像的存储知识，也有助于 Docker 其他概念的理解，如 docker commit、docker build、docker run 等。



Chapter 11

第 11 章
docker build 实现

11.1 引言

Docker 镜像在 Docker 生态圈中起到的作用非同一般。一起来看 Docker 的发展，我们甚至可以如此评价：Docker 正是凭借着其灵活的镜像技术，以及革命性的镜像托管服务 Docker Hub，在云计算时代占据了行业内极为有利的位置。其自身的 Dockerfile 技术甚至有可能取代传统软件发布的模式，成为行业内更快捷、更有效、更易于部署与管理的软件发布模式。

Docker 诞生之前，软件的发布模式一直以代码为核心。软件开发者首先在开发环境中开发软件，开发到一定阶段后，开发者提交代码至托管平台；软件测试者接着在测试环境中部署软件代码，做相应的测试工作并提交测试报告；软件开发者与测试者将软件完善至一定阶段后交付，交付形式一般仍旧是代码。最终交付的软件，仍然需要在另外的环境中部署，因此软件对部署环境的要求需要通过软件说明书的形式交付给客户。然而，真实的部署环境并不一定能与软件有效地结合，这也是以代码为核心的软件发布模式一直以来的痛点。

Docker 诞生之后，软件发布模式似乎有了新的转机。软件代码依然是软件的重要组成部分，然而，与以往不同的是，Docker 生态圈中软件的发布使得代码与运行环境并不分离，而是将软件运行环境也圈入软件范畴，一并交付给客户。客户在获得软件之后，并不需要额外地配置环境，而是能够直接运行软件。这种新模式的诞生极大地吸引了开发者的眼球，传统软件发布弊端的剔除，也逐渐使得 Docker 阵营越来越庞大。

软件发布模式的革新，并非信手拈来，革新理念的外表下是前卫的创新意识，以及创新

意识下的技术积淀。Docker 的镜像技术，可以说是传统联合文件系统在云时代迎来的又一个春天。在镜像技术之上，Dockerfile 的设计则可以认为是 Docker 公司在镜像技术上抽象出的一个全新软件标准。此标准一出，软件开发者有能力以一个超乎想象的速度，发布包含运行环境的软件，并通过 Docker Hub 的方式串联全世界。

第 8 章、第 9 章以及第 10 章分别介绍了 Docker 镜像的原理、Docker 镜像的下载以及 Docker 镜像的存储。在这些镜像技术的基础之上，理解在 Docker 环境中如何通过 Dockerfile 构建自己的 Docker 镜像，就显得意义重大。

本章从 Docker 1.2.0 的源码出发，分析用户通过 Dockerfile 构建出一个 Docker 镜像的来龙去脉。

分析之前，我们首先来简单了解 docker build 的作用。简单而言，用户可以通过一个自定义的 Dockerfile 文件以及相关内容，从一个基础镜像起步，对于 Dockerfile 中的每一条命令，都在原先的镜像 layer 之上再额外构建一个新的镜像 layer，直至构建出用户所需的镜像。

由于 docker build 命令由 Docker 用户发起，故 docker build 的流程会贯穿 Docker Client、Docker Server 以及 Docker Daemon 这三个重要的 Docker 模块。本章也正是以这三个 Docker 模块为主题，分析 docker build 命令的执行，其中 Docker Daemon 作为 Docker 的重中之重，本章对其的分析也将会更加详细。具体而言，本章内容包含以下 3 部分：

1）概述 docker build 命令执行的流程，包含 Docker Client、Docker Server 以及 Docker Daemon；

2）详细分析 Docker Daemon 对 Dockerfile 的 build 实现；

3）简要分析 Dockerfile 中的命令在 Docker Daemon 中的执行流程。

11.2　docker build 执行流程

docker build 可以帮助 Docker 用户通过 Dockerfile 的形式构建出自己的 Docker 镜像。更为细致的描述是：用户通过 Docker Client 向 Docker Server 发送一条 docker build 命令，发送命令时，用户需要首先指定 Dockerfile 以及相关内容，并通过 Docker Client 将请求发出；Docker Server 接收请求之后，将其路由转发至相应的处理方法；Docker Daemon 负责执行请求处理的处理方法，解析 Dockerfile 并构建出最终的镜像。

Docker Client、Docker Server 以及 Docker Daemon 协同完成 build 任务的流程图如图 11-1 所示。

下面几节将从 Docker Client、Docker Server 以及 Docker Daemon 三个方面分析 docker build 的流程。

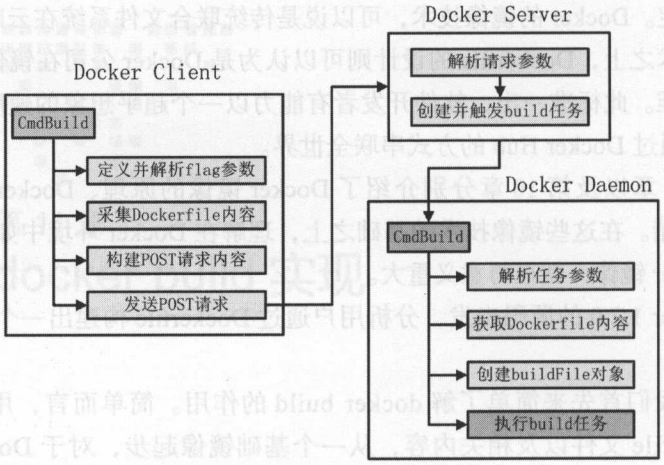


图 11-1　docker build 的流程图

11.2.1　Docker Client 与 docker build

Docker Client 作为用户请求的入口，自然第一个接收并处理 docker build 命令。图 11-1 中也已经清楚地标明 Docker Client 的处理流程。本章基于 Docker 1.2.0 版本进行分析，随着 Docker 版本的不断更新，Docker 在 build 的实现上也有了较为明显的更新，然而，万变不离其宗，docker build 的思想与精髓并未产生本质的变化。Docker Client 处理 docker build 命令的流程图如图 11-2 所示。

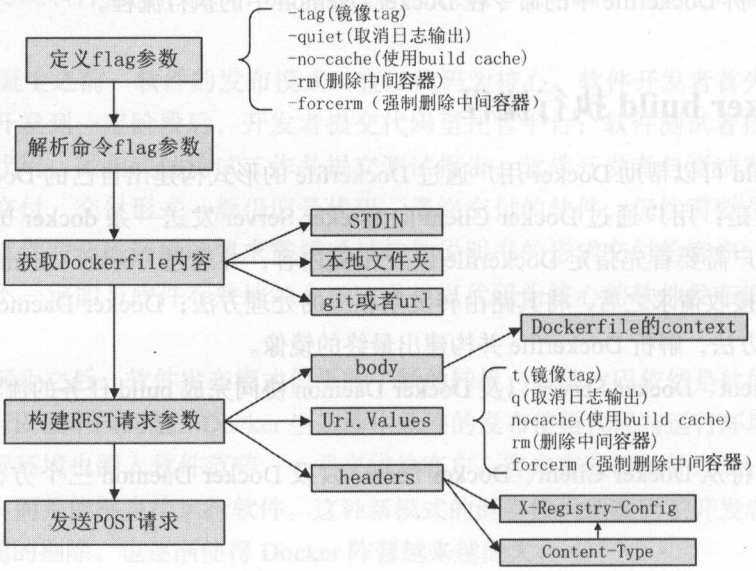


图 11-2　Docker Client 处理 docker build 命令的流程图

关于 Docker Client 端的处理，下面将从 4 个方面分析 docker build 命令的处理流程。关于 Docker 架构中 Docker Client 的构建与命令执行，可以参考第 2 章。

1. 定义并解析 flag 参数

用户发起的 Docker 命令中，很多情况下都会带 flag 参数。一般情况下，Docker Client 会通过定义与解析，对这些 flag 参数进行转义，从而将用户的请求按需转换，使其适应于 Docker Server 暴露的 API 接口。

命令 docker build 的 flag 参数共有 5 个，分别为 tag、suppressOutput、noCache、rm 与 forceRm。flag 参数的源码定义位于 ./docker/docker/api/client/command.go#L102-L106，如下：

```
tag := cmd.String([]string{"t", "-tag"}, "", "Repository name (and optionally a
    tag) to be applied to the resulting image in case of success")
suppressOutput := cmd.Bool([]string{"q", "-quiet"}, false, "Suppress the
    verbose output generated by the containers")
noCache := cmd.Bool([]string{"#no-cache", "-no-cache"}, false, "Do not use
    cache when building the image")
rm := cmd.Bool([]string{"#rm", "-rm"}, true, "Remove intermediate containers
    after a successful build")
forceRm := cmd.Bool([]string{"-force-rm"}, false, "Always remove intermediate
    containers, even after unsuccessful builds")
```

5 个 flag 参数的具体说明参见表 11-1。

表 11-1　docker build 命令的 flag 参数说明

flag 参数	flag 参数形式	flag 参数默认值	flag 参数描述
tag	t、-tag	""（空字符串）	build 完毕后为新镜像设置的 tag 信息
suppressOutput	q、-quiet	false	是否输出容器产生的日志等信息
noCache	-no-cache	false	构建镜像时不使用镜像缓存
rm	-rm	true	成功构建镜像之后，删除中间容器
forceRm	-force-rm	false	强制删除中间容器

需要强调的是：在实际应用场景中，noCache 参数的作用非常神奇。一旦 -no-cache 参数为 false，则说明不使用镜像缓存的定论不成立，也就是说，使用镜像缓存。命令 docker build 命令执行时，镜像缓存会大大缩短相似 Dockerfile 的 build 时间。比如：Dockerfile1 中第 4 条命令为编译、安装某一款软件，执行该命令本身需要占用的时间极长；若 Dockerfile2 中的前 4 条命令与 Dockerfile1 完全一致，并且不涉及内容复制，则 Dockerfile2 在构建前 4 条命令时，完全可以使用 Dockerfile1 构建时的镜像缓存，从而大大压缩了本次 docker build 所消耗的时间。

定义 flag 参数完毕之后，Docker Client 随即解析命令中的 flag 参数，并对处理结果进行处理，源码实现如下：

```
if err := cmd.Parse(args); err != nil {
        return nil
```

```
    }
    if cmd.NArg() != 1 {
        cmd.Usage()
        return nil
    }
```

至此，第一个步骤已经完成，flag 参数解析之后，Docker Client 将立即进入 Dockerfile 内容采集阶段。

2. 获取 Dockerfile 相关内容

命令 docker build 的初衷是为 Docker 用户构建预期的镜像。为了达到这个目的，Docker 用户自然需要向 Docker Daemon 提供必需的原材料。而这些原材料正是 Dockerfile 以及其相关内容。既然如此，Docker Client 则必须代表 Docker 用户提供这些原材料。

Docker Client 在 flag 参数解析完成之后，下一步骤即为获取 Dockerfile 相关内容。最为常见的 Dockerfile 一般为一个文件，存在于 Docker Client 所在的文件系统中，并且伴随着一些其他文件，一般是 Dockerfile 所在目录下的其他相关文件内容。用户往往可以在 Dockerfile 所在目录下执行 docker build 命令来完成镜像构建。这是 Docker 最为普遍的构建方式，分析 build 的源码实现，大家就可以发现：Docker 的 build 命令支持的 Dockerfile 源比指定目录要丰富得多。除此之外，Docker 还支持从 STDIN 读取 Dockerfile、从远程 URL 获取 Dockerfile，以及从 git 源获取 Dockerfile。如何解析 Dockerfile 源并且构建 context 信息的流程图如图 11-3 所示。

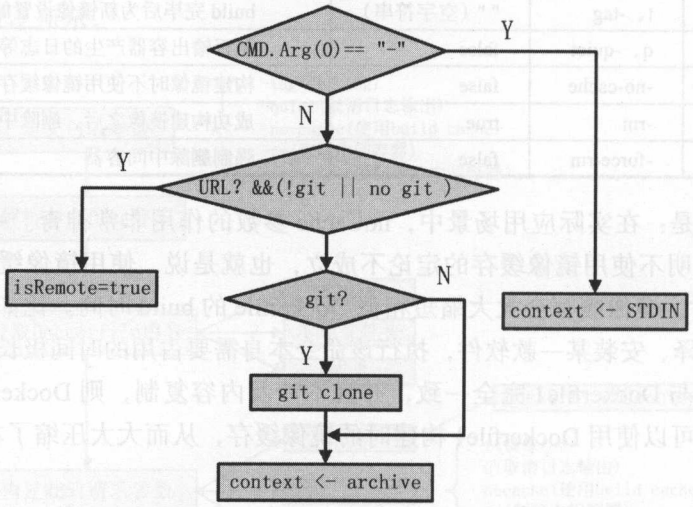


图 11-3　解析 Dockerfile 源并构建 context 信息的流程图

Docker Client 解析 Dockerfile 源的源码位于 ./docker/docker/api/client/command.go#L123-L193，如下：

```
if cmd.Arg(0) == "-" {
    ......
    if !archive.IsArchive(magic) {
        dockerfile, err := ioutil.ReadAll(buf)
        if err != nil {
            return fmt.Errorf("failed to read Dockerfile from STDIN: %v", err)
        }
        context, err = archive.Generate("Dockerfile", string(dockerfile))
    } else {
        context = ioutil.NopCloser(buf)
    }
} else if utils.IsURL(cmd.Arg(0)) && (!utils.IsGIT(cmd.Arg(0)) || !hasGit) {
    isRemote = true
} else {
    root := cmd.Arg(0)
    if utils.IsGIT(root) {
        ......
        if output, err := exec.Command("git", "clone", "--recursive",
        remoteURL, root).CombinedOutput(); err != nil {
            return fmt.Errorf("Error trying to use git: %s (%s)", err, output)
        }
    }
    ......
    options := &archive.TarOptions{
        Compression: archive.Uncompressed,
        Excludes:    excludes,
    }
    context, err = archive.TarWithOptions(root, options)
    if err != nil {
        return err
    }
}
```

简要分析该流程。Docker Client 解析完 flag 参数之后，通过第一个参数 cmd.Arg(0) 来确定 Dockerfile 源。流程大致如下：

1）若该参数为 "-"，则代表 Docker 用户指定 STDIN 作为 Dockerfile 的输入源，Docker Client 随即通过 docker build 命令的标准输入读入数据，压缩后最终得到结果 context；

2）若该参数与 URL 匹配，并且不为 git 地址，又或者参数与 URL 匹配，同时也是 git 地址，但是 Docker Client 所在宿主机没有安装 git，则标记 isRemote 参数为 true，表明 Dockerfile 需要远程获取。

3）若该参数为 git 地址，且本地安装 git，则 Docker Client 首先将 git 内容通过 git clone 命令复制到本地；否则，表明 Dockerfile 已经位于当前目录，Docker Client 解析当前路径之后，将内容压缩打包，得到最终结果 context。

Docker Daemon 执行 docker build 命令的原材料已经辨析清楚、准备完毕。此时，Docker Client 需要为发送请求给 Docker Daemon 做准备。

3. 构建 REST 请求参数

构建 REST 请求，标志着 Docker Client 与 Docker Daemon 建立通信的开始。Docker Client 需要为该请求配置相应的参数，比如请求体，请求 url 中的参数值，以及请求 header 等。

请求体参数的配置，源码位于 ./docker/docker/api/cli/command.go#L194-L200，如下：

```
var body io.Reader
// Setup an upload progress bar
// FIXME: ProgressReader shouldn't be this annoying to use
if context != nil {
    sf := utils.NewStreamFormatter(false)
    body = utils.ProgressReader(context, 0, cli.err, sf, true, "", "Sending
        build context to Docker daemon")
}
```

简单而言，请求体的配置依赖 context 的内容，即 Dockerfile 以及相关内容将作为请求的 body。

请求 url 的参数配置紧随其后，这些 url 参数有 tag、quiet、remote、nocache、rm 与 forcerm。除了 remote 参数的值来源于 cmd.Arg(0) 之外，其他参数均来源于 docker build 命令的 flag 参数。

请求参数配置的最后阶段是请求 header。由于 docker build 请求同样有可能需要用户的认证信息，故 Docker Client 为请求添加了名为 X-Registry-Config 的 header，值为用户的 Config 信息。同时也会由于请求 body 的类型，而设置 header 中的 Content-Type 参数。

4. 发送 POST 请求

Docker Client 万事准备就绪之后，最后一个步骤即发送 POST 请求至 Docker Server，由 Docker Server 将请求路由至 Docker Daemon 中具体的处理方法。

发送请求的源码位于 ./docker/docker/api/cli/command.go#L245，如下：

```
err = cli.stream("POST", fmt.Sprintf("/build?%s", v.Encode()), body, cli.out, headers)
```

请求类型为 POST，请求的 URL 为 /build，并携带 url 的查询参数值，请求体为 body，请求也带有 header 信息。

当 POST 请求发送完毕之后，docker build 在 Docker Client 的处理就基本完成了，接着由 Docker Server 以及 Docker Daemon 扮演处理请求的角色。

11.2.2 Docker Server 与 docker build

正如 Docker Server 一如既往的角色，它负责根据请求类型以及请求的 URL，路由转发 Docker 请求至相应的处理方法。在处理方法中，Docker Server 会创建相应的 Job，为 Job 配置相应的执行参数并触发该 Job 的运行。

命令 docker build 在 Docker Client 中最终通过一个 POST 请求发出，URL 前缀为 build。而 Docker Server 的路由表中有以下这条规则，位于 ./docker/docker/api/server/server/

go#L1123，如下：

```
"POST": {
    ...
    "/build":                    postBuild,
    ...
}
```

因此对于 docker build 请求，Docker Server 执行的处理方法为 postBuild。postBuild 的定义与实现位于 ./docker/docker/api/server/server.go#L913，如下：

```
func postBuild(eng *engine.Engine, version version.Version, w http.
            ResponseWriter, r *http.Request, vars map[string]string) error
```

此处理方法的职责与其他处理方法无异：解析请求参数，创建并触发执行 Job。创建的 Job 名为 build，代码如下：

```
job = eng.Job("build")
```

而 Docker Client 传递至 Docker Server 的 url 参数，均会按规则对 job 的环境变量赋值。需要特殊说明的是：关于 POST 请求的 body，也就是 Dockerfile 相关内容，会以 Job 自身标准输入的形式添加至 job 内部，代码如下：

```
job.Stdin.Add(r.Body)
```

所有环境变量配置完毕之后，Docker Server 执行 job.Run，触发执行这个名为 build 的 Job。

11.2.3　Docker Daemon 与 docker build

Docker Daemon 作为 Docker 体系中的“大脑”部分，任务真正的执行都由它来完成，请求 docker build 也不例外。处理 docker build 请求，意味着 Docker Daemon 会接管 build 这个 Job 的执行权，并将任务完成。

Docker Daemon 开始执行的任务无外乎也是解析 Job 环境变量，获取 Dockerfile 内容。解析 Job 环境变量的源码位于 ./docker/docker/daemon/build.go#L40-L53，如下：

```
var (
    remoteURL      = job.Getenv("remote")
    repoName       = job.Getenv("t")
    suppressOutput = job.GetenvBool("q")
    noCache        = job.GetenvBool("nocache")
    rm             = job.GetenvBool("rm")
    forceRm        = job.GetenvBool("forcerm")
    authConfig     = &registry.AuthConfig{}
    configFile     = &registry.ConfigFile{}
    tag            string
    context        io.ReadCloser
)
```

```
job.GetenvJson("authConfig", authConfig)
job.GetenvJson("configFile", configFile)
```

Dockerfile 内容的获取紧随其后。Dockerfile 及其相关内容可以通过三种源向 Docker Daemon 交付，因此，Docker Daemon 获取相关内容时，也需要首先通过参数分辨是哪种具体方式，并实现内容的最终提取。Docker Daemon 获取 Dockerfile 相关内容 context 的流程图如图 11-4 所示。

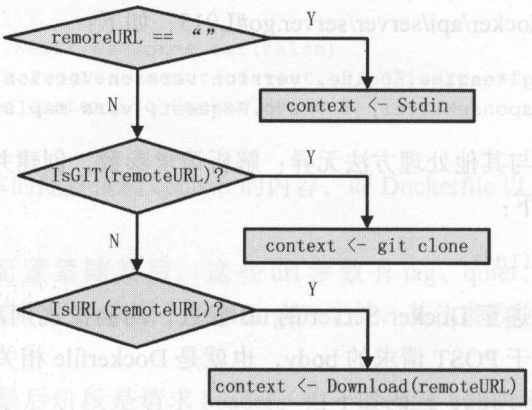


图 11-4 Docker Daemon 获取 context 的流程图

这部分的源码实现简单明了，位于 ./docker/docker/daemon/build.go#L56-L92，如下：

```
if remoteURL == "" {
    context = ioutil.NopCloser(job.Stdin)
} else if utils.IsGIT(remoteURL) {
    if !strings.HasPrefix(remoteURL, "git://") {
        remoteURL = "https://" + remoteURL
    }
    root, err := ioutil.TempDir("", "docker-build-git")
    if err != nil {
        return job.Error(err)
    }
    defer os.RemoveAll(root)

    if output, err := exec.Command("git", "clone", "--recursive", remoteURL,
                root).CombinedOutput(); err != nil {
        return job.Errorf("Error trying to use git: %s (%s)", err, output)
    }

    c, err := archive.Tar(root, archive.Uncompressed)
    if err != nil {
        return job.Error(err)
    }
    context = c
} else if utils.IsURL(remoteURL) {
```

```
        f, err := utils.Download(remoteURL)
        if err != nil {
            return job.Error(err)
        }
        defer f.Body.Close()
        dockerFile, err := ioutil.ReadAll(f.Body)
        if err != nil {
            return job.Error(err)
        }
        c, err := archive.Generate("Dockerfile", string(dockerFile))
        if err != nil {
            return job.Error(err)
        }
        context = c
    }
```

分析以上源码，我们可以发现 Docker Daemon 以下的处理逻辑。若 context 成功获取，则说明构建 Docker 镜像的原材料已经准备完毕，Docker Daemon 接着将逐一分析 context 中 Dockerfile 的内容，并完成相应的 build 语句。虽然构建 Docker 镜像的原材料已经完备，但是由 Docker Daemon 的哪个部件来完成这项重大的工作，仍然未给出答案。紧接着，Docker Daemon 根据 context 获取的源代码，创建一个 buildFile 对象。命令 docker build 的生命周期中，buildFile 起到的作用非同小可。Dockerfile 中书写的所有命令均由 buildFile 来完成相应的 build 操作。

首先，我们进入 buildFile 结构体的定义：

```
type buildFile struct {
    daemon *Daemon
    eng    *engine.Engine
    image      string
    maintainer string
    config     *runconfig.Config
    contextPath string
    context    *tarsum.TarSum
    verbose     bool
    utilizeCache bool
    rm          bool
    forceRm     bool
    authConfig *registry.AuthConfig
    configFile *registry.ConfigFile
    tmpContainers map[string]struct{}
    tmpImages     map[string]struct{}
    outStream io.Writer
    errStream io.Writer
    // Deprecated, original writer used for ImagePull. To be removed.
    outOld io.Writer
    sf     *utils.StreamFormatter
    // cmdSet indicates is CMD was set in current Dockerfile
    cmdSet bool
}
```

对于 buildFile 结构体，部分属性的含义如表 11-2 所示。

表 11-2 buildFile 属性说明

属性名称	含义	属性名称	含义
daemon	Job 所属 Daemon	forceRm	强制删除中间容器
engine	Job 所属 Engine	authConfig	认证信息
image	基础镜像	configFile	配置文件
maintainer	Dockerfile 维护者	tmpContainer	临时容器列表
config	运行配置参数	tmpImages	临时镜像列表
contextPath	context 所在路径	outStream	输出流
context	context 内容	errStream	错误流
verbose	输出 build	outOld	Job 的标准输出
utilizeCache	使用镜像缓存	sf	StreamFormatter
rm	删除中间容器	cmdSet	是否指定 cmd

其中，buildFile 可以认为是一个生产镜像车间，只要有原材料（Dockerfile）输入，它就可以按照要求为用户生产 Docker 镜像。Docker 构建并初始化 buildFile 对象之后，随后即开始真正的 build 之旅，车间开始运作，按要求构建镜像。镜像一旦构建成功，Docker Daemon 将镜像 ID 在 Repository 中注册，以便后续使用。这部分的源码位于 ./docker/docker/daemon/build.go#L106-L112，如下：

```
id, err := b.Build(context)
if err != nil {
    return job.Error(err)
}
if repoName != "" {
    daemon.Repositories().Set(repoName, tag, id, false)
}
```

构建与注册两项工作的完成，代表 Docker Daemon 的 build 任务大功告成。Docker Daemon 响应 Docker Server，并返回请求响应至 Docker Client，通知用户镜像构建任务的完成情况。

Docker Client、Docker Server 以及 Docker Daemon 三者协同完成命令 docker build 的流程已经分析完毕。目前，我们清晰的是流程，模糊的是具体的 build 执行。11.3 节将给出代码 b.Build(context)，分析 Dockerfile 中具体命令的执行细节。

11.3 Dockerfile 命令解析流程

Docker Server 交付给 Docker Daemon 的内容是一系列参数，以及 Dockerfile 相关内容压缩后的 context 对象，我们习惯于将后者称为原材料。Dockerfile 命令解析也是从原材料入手，以下是 build 函数的执行流程，如图 11-5 所示。

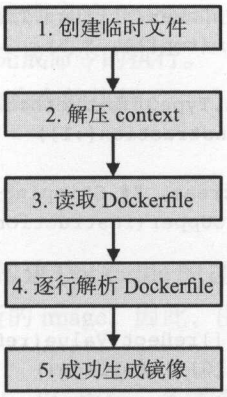


图 11-5　build 函数执行流程图

函数 build 的执行过程中，第 1 ~ 3 步，以及第 5 步的执行内容及意义简单而清晰，本节主要分析第 4 步——逐行解析 Dockerfile。逐行解析 Dockerfile 命令的代码位于 ./docker/docker/daemon/build.go#898-L912，如下：

```
for _, line := range strings.Split(dockerfile, "\n") {
    line = strings.Trim(strings.Replace(line, "\t", " ", -1), " \t\r\n")
    if len(line) == 0 {
        continue
    }
    if err := b.BuildStep(fmt.Sprintf("%d", stepN), line); err != nil {
        if b.forceRm {
            b.clearTmp(b.tmpContainers)
        }
        return "", err
    } else if b.rm {
        b.clearTmp(b.tmpContainers)
    }
    stepN += 1
}
```

以上代码的 for 循环中，每一个循环均会传入 Dockerfile 中的一行，代码中变量为 line。预处理 line 之后，每次循环执行的任务是 b.BuildStep() 函数，并在每一个循环的最后，对循环次数进行统计。BuildStep 函数的作用是从 line 中解析相应的 Dockerfile 指令，完成构建一个镜像 layer 的任务，并在当前上下文中执行。BuildStep 函数的定义位于 ./docker/docker/daemon/build.go#L921-L943，如下：

```
func (b *buildFile) BuildStep(name, expression string) error {
    fmt.Fprintf(b.outStream, "Step %s : %s\n", name, expression)
    tmp := strings.SplitN(expression, " ", 2)
    if len(tmp) != 2 {
        return fmt.Errorf("Invalid Dockerfile format")
    }
```

```
    instruction := strings.ToLower(strings.Trim(tmp[0], " "))
    arguments := strings.Trim(tmp[1], " ")

    method, exists := reflect.TypeOf(b).MethodByName("Cmd" + strings.
                ToUpper(instruction[:1]) + strings.ToLower(instruction[1:]))
    if !exists {
        fmt.Fprintf(b.errStream, "# Skipping unknown instruction %s\n",
                strings.ToUpper(instruction))
        return nil
    }

    ret := method.Func.Call([]reflect.Value{reflect.ValueOf(b), reflect.
        ValueOf(arguments)})[0].Interface()
    if ret != nil {
        return ret.(error)
    }

    fmt.Fprintf(b.outStream, " ---> %s\n", utils.TruncateID(b.image))
    return nil
}
```

自行编写过 Dockerfile 的 Docker 爱好者对于内部每一行的书写肯定非常清楚。Dockerfile 的每一行都必须由命令类型和命令参数两部分构成，如：

```
FROM ubuntu:14.04
MAINTAINER Allen Sun allen.sun@daocloud.io
RUN apt-get update
CMD [ “/bin/bash” ]
```

因此，BuildStep 首先通过一行中第一个空格字符 " " 将传入内容分离，如下：

```
tmp := strings.SplitN(expression, " ", 2)
instruction := strings.ToLower(strings.Trim(tmp[0], " "))
arguments := strings.Trim(tmp[1], " ")
```

数组中，tmp[0] 代表 Dockerfile 中相应行的命令类型，tmp[1] 代表命令参数。两者解析完毕后，分别赋值给 instruction 与 arguments。命令类型获取之后，Docker Daemon 巧妙地使用了 Golang 中的反射（reflect）机制获取具体的执行方法。执行方法提取后，执行时传入命令参数，即完成了一条 Dockerfile 指令的执行，代码如下：

```
method, exists := reflect.TypeOf(b).MethodByName("Cmd" + strings.
            ToUpper(instruction[:1]) + strings.ToLower(instruction[1:]))
ret := method.Func.Call([]reflect.Value{reflect.ValueOf(b), reflect.
            ValueOf(arguments)})[0].Interface()
```

对于 Dockerfile 内的每一条命令，Docker Daemon 都会执行一次循环并通过反射完成方法的执行。如 Dockerfile 中的命令 FROM ubuntu:14.04，首先可以解析出命令类型为 FROM，命令参数为 ubuntu:14.04，即 instruction 值为 from，arguments 为 ubuntu:14.04；接着反射

机制将通过字符串“CmdFrom”找到方法 CmdFrom，即 method 为 CmdFrom；最后通过 method.Func.Call() 函数传入参数，完成命令的执行。

11.4 节将分析 Dockerfile 内具体命令的执行。

11.4　Dockerfile 命令分析

众所周知，Docker Daemon 在构建 Dockerfile 的过程中，对 Dockerfile 中的每一条命令（FROM 命令除外）都会构建一个新的 image。因此，围绕 Docker 的镜像技术，build 的流程也是创建一个个 image 的过程。

对于一个 Dockerfile，或者 Dockerfile 内的一条或多条命令，buildFile 对象均完美地记录所有命令执行时的上下文现状。

Dockerfile 内部的命令简单，可以分为两类。第一类命令修改上一层 image 的文件系统内容，比如：命令 RUN 在基于上一层 image 的容器中运行一条指令，对于用户而言，该指令很有可能修改上一层 image 的内容；命令 ADD 在 Dockerfile 所在目录的 context 中复制内容至上一层 image，用户视角下同样属于修改镜像；这样的命令还有 COPY 等。第二类命令仅仅修改镜像的 config 信息，比如：命令 ENV 不会修改镜像文件系统中的内容，而仅仅修改镜像的 config 的 ENV 信息，以便后续使用该镜像启动进程时，ENV 信息作为进程的环境变量加载；同样 EXPOSE 命令代表以该镜像运行容器时，容器内进程会监听 EXPOSE 的端口号，以便 Docker Daemon 捕获容器内的端口监听情况，这部分信息同样会被记录至 config 信息，最后被更新至镜像的 json 文件中；第二类命令还包括很多，如 CMD、ENTRYPOINT、MAINTAINER 等。

本节着重分析三种具有代表性的 Dockerfile 命令 FROM、RUN、ENV，阐述这些命令解析执行时，构建 Docker 新镜像的详细过程。

11.4.1　FROM 命令

FROM 命令一般都是 Dockerfile 中的首条命令，紧跟其后的参数为具体的镜像名称，意味着整个 build 的流程都将此镜像作为基础镜像。虽然作为 build 流程的基础镜像，但是依旧没有谈及实现方式。此时，buildFile 这个对象就体现了重要的作用，很多命令的结果都会在 buildFile 对象中有所体现。在这里，FROM 命令的基础镜像信息会被记录到 buildFile 对象中，其中与 image 对象对应的属性值为 buildFile.image。

FROM 命令的执行流程图如图 11-6 所示。

从图 11-6 中，我们可以发现对于 FROM 命令后的镜像参数，Docker Daemon 首先在 daemon.repository 中查找指定的镜像，若镜像不存在，即立即执行镜像下载任务，Docker 镜像的下载可以参见第 9 章；若镜像存在，则获取镜像信息，并开始 buildFile 配置流程。配置 buildFile 的作用是将镜像的信息加载至 buildFile。关于 FROM 命令的解析工作，源码位于

./docker/docker/daemon.go#L171-L233，如下：

```
b.image = image.ID
b.config = &runconfig.Config{}
if image.Config != nil {
    b.config = image.Config
}
if b.config.Env == nil || len(b.config.Env) == 0 {
    b.config.Env = append(b.config.Env, "PATH="+DefaultPathEnv)
}
```

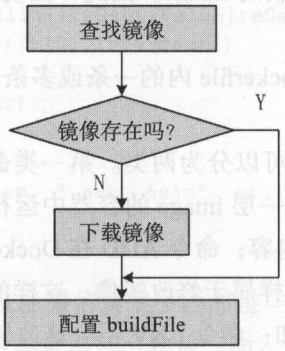


图 11-6 FROM 命令执行流程图

分析以上源码可以发现，Docker Daemon 首先获取镜像 ID，将值传给 buildFile 的 image 属性。如此一来，build 流程的基础镜像信息已经获取，Dockerfile 的第二条命令可以在此基础镜像的基础上来完成。当然，buildFile 的 image 属性也会随着 build 流程的变化而变化。举一个最简单的例子，第三条命令就会依赖第二条命令构建完之后的 image。回到 buildFile 的配置，以上代码表明若镜像 config 信息存在，则将镜像的 config 信息传递至 buildFile 的 config 属性；而 config 属性中的 Env 属性若为空，则在 Env 属性中添加默认 PATH，常量为 DefaultPathEnv，如下：

```
const DefaultPathEnv = "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin"
```

CmdFrom 执行的最后环节是配置 buildFile 的 OnBuild 属性。

总之，CmdFrom 完成基础镜像的加载，通过 buildFile 对象初始化 build 流程所需的配置信息。

11.4.2 RUN 命令

RUN 命令在 build 流程中扮演着非常重要的角色，它允许在镜像的基础上执行用户指定的命令。RUN 不同于其他仅修改镜像 config 信息的命令，前者执行用户指定的命令，这意味着需要在镜像基础上执行动态的命令，执行命令时存在“容器”的概念。

RUN 命令的执行通过 CmdRun 函数来完成，源码实现位于 ./docker/docker/daemon/build.go#L276-L320，简化后如下：

```
func (b *buildFile) CmdRun(args string) error {

    ......
    b.config.Cmd = config.Cmd

    hit, err := b.probeCache()
    if err != nil {
        return err
    }
    if hit {
        return nil
    }

    c, err := b.create()
    ......
    c.Mount()
    ......
    err = b.run(c)
    ......
    if err := b.commit(c.ID, cmd, "run"); err != nil {
        return err
    }

    return nil
}
```

分析以上源码，可以归纳总结出 CmdRun 函数的执行流程图，如图 11-7 所示。

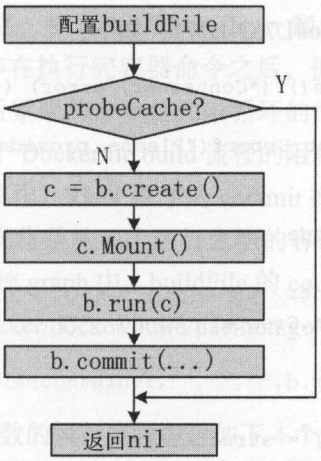


图 11-7　CmdRun 函数的执行流程图

CmdRun 函数的执行流程图，可谓步步精要，每一个步骤在 Docker 架构中都是一个非常

重要的概念。以下从 probeCache、b.create()、c.Mount()、b.run(c) 以及 b.commit() 这 5 个方面进行分析。

1. 镜像 cache 机制

首先分析 probeCache，一起来领略 Docker 在 build 流程中巧妙使用镜像 cache 的精彩。镜像 cache 的重用意味着构建相应的命令时，不再执行 RUN 命令后的整条命令，而从 Docker Daemon 本地的镜像中找出效果与执行该命令相同的镜像，作为结果返回。

实现原理也不难理解。由于 Docker Daemon 在 build RUN 命令之前，buildFile 的属性 image 肯定有一个值（基础镜像的 ID 或者后续覆盖基础镜像的镜像 ID）；又由于在执行 build RUN 命令时，Docker Daemon 首先需要配置 buildFile 的 config 属性，实则是在 buildFile.image 的 config 属性上进行配置。因此，Docker Daemon 只需遍历本地所有镜像，只要存在一个镜像，此镜像的父镜像 ID 与当前 buildFile 的 image 值相等，同时此镜像的 config 内容与 buildFile.config 相同，则完全可以认为执行 RUN 命令产生的结果与此镜像的效果一致，直接使用本地存储的此镜像即可，无须额外再创建。若不存在满足条件的镜像，则说明将要构建的新镜像在本地还不存在，Docker Daemon 必须完整执行 RUN 命令。

2. 创建 Container 对象

不能使用镜像 Cache 机制，意味 Docker Daemon 没有捷径可走，随后的流程是创建 Container 对象，为运行容器做准备。

运行 Docker 容器之前，Docker Daemon 会为其创建一个 Container 对象，RUN 命令执行流程中 `c,err := b.create()` 即创建了一个 Container 类型的实例 c。在 Docker Daemon 的范畴内，创建一个 Container 对象，只需要基础镜像的 ID，以及运行容器时所需的 runconfig 信息，而这些信息在 build 流程中都被 buildFile 统一管理。函数 create 的定义位于 ./docker/docker/daemon/build.go#L752-L771，如下：

```
func (b *buildFile) create() (*Container, error) {
    if b.image == "" {
        return nil, fmt.Errorf("Please provide a source image with 'from'
        prior to run")
    }
    b.config.Image = b.image

    // Create the container
    c, _, err := b.daemon.Create(b.config, "")
    if err != nil {
        return nil, err
    }
    b.tmpContainers[c.ID] = struct{}{}
    fmt.Fprintf(b.outStream, " ---> Running in %s\n", utils.TruncateID(c.ID))

    // override the entry point that may have been picked up from the base image
    c.Path = b.config.Cmd[0]
```

```
    c.Args = b.config.Cmd[1:]

    return c, nil
}
```

以上代码中，涉及的 config 参数有当前镜像的 ID、所有的 buildFile config 信息以及 config 中的 Cmd 信息（包括 Cmd 的路径以及 Cmd 的参数）。

3. 挂载文件系统

RUN 命令需要在容器中运行指定的程序，故仅仅创建 Container 类型实例 c 还不足以支撑容器的运行。CmdRun 仍然需要为容器的运行挂载文件系统，c.Mount() 即实现了这部分功能。由于 Container 类型实例 c 中包含镜像的 ID，因此 Docker Daemon 根据该镜像 ID，追溯出此 ID 的所有祖先镜像，并将所有镜像通过指定的 graphdriver 完全联合起来，挂载到同一个目录下。而后容器的运行将使用该挂载点，作为容器的根目录。

4. 运行容器

运行 Docker 容器，绝对是 Docker 体系中的重头戏。如果容器缺少了动态的运行时，对于用户而言，Docker 就显得没有丝毫吸引力。

为容器的运行创建了 Container 类型实例 c 之后，CmdRun 即利用 c 中众多的容器配置信息，将 Docker 容器运行起来。在这一过程中，Docker Daemon 完成的工作有很多，包括：创建容器的文件系统，创建容器的命名空间进行做资源隔离，为容器配置 cgroups 参数进行资源控制，当然，还有运行用户指定的程序等。运行容器举足轻重，这部分内容将在第 12 章详细展开。

5. 提交新镜像

运行完容器，DockerDaemon 需要对运行后的容器进行 commit 操作，将容器运行的结果保存在一个新的镜像中，换言之，将更改后的 top layer 制作成一个新镜像，并有效存储。需要注意的是，虽然 commit 操作在执行完容器命令之后，但是如何评价一个命令执行完却有不少难度。一旦 RUN 命令之后紧跟的命令是无限循环的命令，或者不会退出的命令，容器运行就不会退出，这将导致整个 Dockerfile build 流程的阻塞。

回到 buildFile 的 commit 操作，RUN 命令的 commit 操作和其他命令的 commit 操作稍有不同。RUN 命令的 commit 操作是从一个运行完毕的容器中保存文件系统中的 Read-Write 层，以一个镜像的形式存入本地 graph 中。buildFile 的 commit 函数直接调用 daemon 包中的 Commit 函数，源代码位于 ./docker/docker/build/daemon.go#853，如下：

```
image,err:=b.daemon.Commit(container,"","","",b.maintainer,true,&autoConfig)
```

而 daemon 包的 Commit 函数的执行流程包含如下 4 个步骤。

1）暂停 Docker Container 的运行。这一步骤对于 Dockerfile 中的 RUN 命令不起作用，原因是 CmdRun 命令只有在命令完全执行完毕，Docker 容器终止运行之后，才执行 commit 操作。然而，对于一个正在运行的容器，Docker 同样支持为容器提交一个镜像，而此时

Commit 操作的第一步即为停止容器的运行，保存容器运行的现场。

2）把容器文件系统的 Read-Write 层打成 tar 包。由于在容器的运行过程中所有写操作都只会作用于文件系统中的 top layer（即 Read-Write 层），故容器运行过程中文件系统的增量变化都在 Read-Write 层，将该层打包即可获得新镜像的文件系统内容。

3）创建 image 对象，并在 Graph 中注册。新的 tar 包即为镜像的原材料，通过 tar 包，以及众多配置信息，DockerDaemon 为其创建一个 image 对象。最后通过 graph.Register() 函数实现在 Graph 中注册该镜像。graph.Register() 函数相信大家一定不会陌生，第 10 章已经分析了该函数的实现。

4）在 Docker 的 repositories 中注册新创建的镜像。对象 repositories 的类型实则为 TagStore，TagStore 的 Repositories 属性即存储了 image 的信息，便于用户快速查找。

回到 buildFile 的 CmdRun 函数，执行 commit 操作之后，立即返回创建的新镜像，而将该镜像的 ID 作为下一个 Dockerfile 命令执行的基础镜像，源代码为 b.image=image.ID，同时 CmdRun 函数也执行完毕。

11.4.3 ENV 命令

ENV 命令的含义为：构建镜像时，为镜像添加一个环境变量。ENV 命令可以有效帮助用户自定义 Docker 镜像的环境变量，以至于通过镜像运行容器时，容器进程能拥有用户定义的环境变量。

执行 ENV 命令的函数为 CmdEnv，CmdEnv 的主要工作即为在 buildFile.image 的基础上，配置指定 buildFile.config 中的 Env 参数。随后执行 commit 操作，源码实现如下：

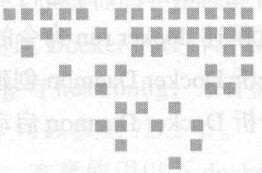
```
b.commit("", b.config.Cmd, fmt.Sprintf("ENV %s", replacedVar))
```

执行以上代码将创建一个新镜像，并运行完 daemon 包中的 Commit 函数。虽然创建镜像过程中不会有新的文件系统变化，但是对于镜像而言，镜像的 json 信息已经发生明显的变化，即镜像的 json 信息中 ENV 部分被修改。

11.5 总结

Dockerfile build 的流程是不断创建新镜像的过程。有的镜像包含文件系统的内容，这意味着镜像创建过程中，容器的 Read-Write 层被修改；有的镜像内容为空，即不包含文件系统的内容，这意味着镜像创建过程中，仅仅修改了镜像的 json 信息。

Dockerfile 的存在，使得软件有能力与运行环境一同以一种非常轻量级的方式分发。不论是软件的发布，还是软件的管理方面，Dockerfile 都大大释放了软件的生命力。本章即从 docker build 的流程入手，从原理的角度分析了构建镜像的全过程。熟悉 docker build 的流程，对于编写高效、合理的 Dockerfile，具有非常大的帮助。



第 12 章 Chapter 12

Docker 容器创建

12.1 引言

云计算时代，随着 Docker 的异军突起，工业界刮过阵阵容器的飓风。飓风所到之处，圈内人士纷纷探讨 Docker 与传统虚拟化技术的异同。传统的虚拟化技术，如 KVM、Xen 等，在过去的数年间已经受到行业的检验，虽然不是尽善尽美，但给工业界带来的好处也足以让世人对其称赞。Docker 的诞生并不是为了替代传统的虚拟化技术，然而，它的诞生却让人如拿着放大镜般地看待传统虚拟化技术的弊病。可以说，Docker 指明了又一条虚拟化道路；或许说 Docker 拓宽了虚拟化的范畴。传统的虚拟化技术主要用于提供基础设施的服务，而 Docker 在隔离与控制计算资源的同时仍然可以融入用户的应用逻辑，基础设施和上层应用的界限被打破。存在即合理，风靡全球更说明了 Docker 满足了用户以往那些被认为天方夜谭的需求。

一直认为 Docker 可以提供“容器”服务，正是 Docker 提供的“容器”会经常用来与“虚拟机”比较。很多比较的维度大家肯定不陌生，比如：“容器”运行时与宿主机共享同一个操作系统，节省物理资源；“容器”的启动非常快，甚至可以达到秒级，“容器”运行时 I/O 性能明显高于虚拟机……工业界对两者的比较肯定比以上罗列的更全面、更具体。然而，似乎很多观点都和“容器”的运行息息相关。而“容器”的运行到底是何种情况？本章将以 Docker 为例，展现 Docker 容器运行的始末。

本章主要从源码的角度分析用户发起 docker run 命令之后，整个 Docker 体系中的组件如何协同工作，最终实现 Docker 容器的运行。本书的分析均基于 Docker 1.2.0 版本，libcontainer 的版本也为 1.2.0。本章主要内容包含以下三个方面：

1）简要讲述 Docker 容器运行的流程，即从 Docker Client、Docker Server 以及 Docker Daemon 的角度阐述 docker run 命令的执行流程；

2）详细分析 Docker Daemon 创建容器对象的过程；

3）详细分析 Docker Daemon 启动容器的过程。

12.2 Docker 容器运行流程

一位精通 Docker 的好手，对于 docker run 命令的使用绝对了然于胸。此命令帮助 Docker 用户在指定配置下完成 Docker 容器的运行，并运行用户指定的程序。Docker 架构中，几乎所有的操作均与 docker run 有莫大的关联。镜像下拉命令 docker pull，目标正是通过 docker run 命令在镜像的基础上运行容器；镜像构建命令 docker build，一旦 Dockerfile 中涉及 RUN 命令，每执行一个 RUN 命令，均会运行一个 Docker 容器，并对容器进行 commit 操作，打包容器镜像；而 Docker 中设置网络模式、容器互联（link）等操作，均需要在容器运行前给 docker run 命令指定参数。

命令 docker run 与其他命令无差别，执行流程均由 Docker Client、Docker Server 以及 Docker Server 协同完成。第 7 章分析 Docker 容器的网络时曾分析过此执行流程，此流程图如图 12-1 所示（Docker Server 仅负责路由转发，图中并未显著标出）：

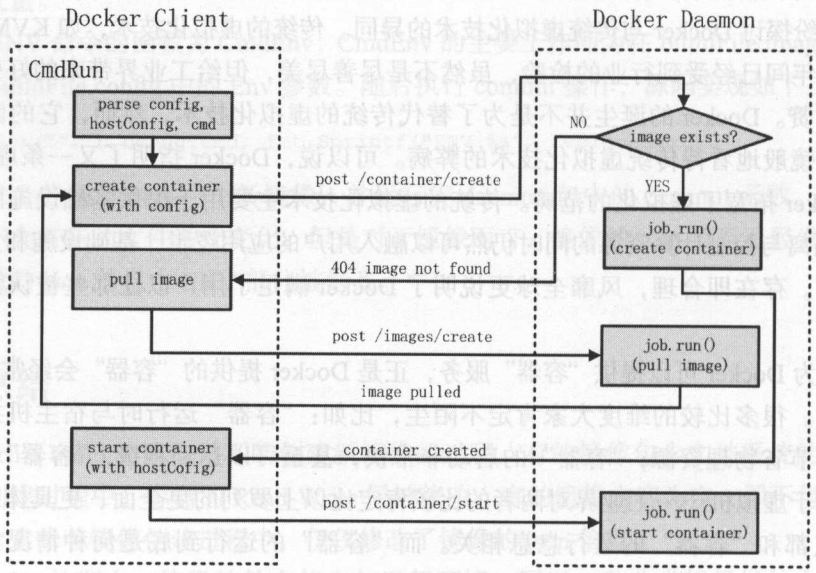


图 12-1 docker run 命令执行流程图

图 12-1 表明，Docker Client 共发送了两次 POST 请求至 Docker Daemon。第一次请求试图让 Docker Daemon 通过请求中的镜像信息，创建容器对象；第二次请求则通知 Docker

Daemon 启动容器，使得容器处于运行状态。对于 Docker Client 而言，docker run 命令的所有参数都需要处理成 Docker Daemon 可以识别的形式，为此 Docker 的设计者设计了 Docker Client 与 Docker Daemon 均可以识别的两个数据结构 config 与 hostconfig，来存储所有命令参数。关于 config 结构体与 hostconfig 结构体的描述与分析，可以参见 7.3.2 节。

为了阐述清楚 config 与 hostconfig 中包含的参数信息，本章使用以下 docker run 命令进行举例分析：

```
docker run -m 104857600 -P --net=bridge -v /home/test:/home/test -v /data -e
TEST_ENV=myenv --name ubuntu_test --link db:db ubuntu:14.04
```

以上命令中参数的分析如表 12-1 所示。

表 12-1　docker run 命令参数解析

参数名称	参数形式	参数含义
-m	-m 104857600	为容器运行设置内存上限 104857600 字节，即 100MB
-P	-P	将容器内 EXPOSE 的端口均暴露至宿主机
--net	--net=bridge	容器运行时使用 bridge（桥接）网络模式
-v	-v /home/test:/home/test	将宿主机目录 /home/test 挂载至容器的 /home/test
-v	/data	为容器创建一个非绑定挂载（bind-mount）的 data volume /data
-e	–e TEST_ENV=myenv	为容器添加环境变量 TEST_ENV，值为 myenv
--name	--name ubuntu_test	为容器设置名称 ubuntu_test
--link	--link db:db	将容器 db 以别名 db 的形式链接到创建的容器中

Docker Client 的参数解析工作是：处理类似以上的参数并存入 config 或者 hostconfig，最后分别通过两次 POST 请求发送至 Docker Daemon。Docker Client 与 Docker Server 解析处理请求的分析，本章已经提及过，故不再赘述。本章仅从 Docker Daemon 两次处理并响应 POST 请求入手，还原 Docker 容器运行的现场。

12.3　Docker Daemon 创建容器对象

Docker 容器的运行并非只是简单地调用内核接口，Docker 的设计者有意将配置信息与启动容器的操作进行区分，实现数据与逻辑的有机分离。

Docker 用户指定的众多参数，以及 Docker 镜像中的众多参数，在运行容器时，均可以认为是 Docker 容器的配置信息来源。当第一次接收到 Docker Client 发送的 POST 请求后，Docker Daemon 开始创建容器对象 container，处理并整理用户指定与镜像指定的 config 信息。创建 container 对象的源码实现位于 ./docker/docker/daemon/create.go#L56-L86，如下：

```
// Create creates a new container from the given configuration with a given name.
func (daemon *Daemon) Create(config *runconfig.Config, name string) (*Container, []
string, error) {
```

```
    var (
        container *Container
        warnings  []string
    )

    img, err := daemon.repositories.LookupImage(config.Image)
    if err != nil {
        return nil, nil, err
    }
    if err := img.CheckDepth(); err != nil {
        return nil, nil, err
    }
    if warnings, err = daemon.mergeAndVerifyConfig(config, img); err != nil {
        return nil, nil, err
    }
    if container, err = daemon.newContainer(name, config, img); err != nil {
        return nil, nil, err
    }
    if err := daemon.createRootfs(container, img); err != nil {
        return nil, nil, err
    }
    if err := container.ToDisk(); err != nil {
        return nil, nil, err
    }
    if err := daemon.Register(container); err != nil {
        return nil, nil, err
    }
    return container, warnings, nil
}
```

Create 函数的逻辑极其清晰，从起初声明 container 对象，到最后返回 container，中间环节的逻辑完全属于顺序执行。表 12-2 给出了中间各环节执行的函数与作用。

表 12-2 Create 函数执行步骤解析

执行函数名称	执行函数作用
LookupImage	在 daemon 对象的 repositories 属性中查找用户指定镜像
CheckDepth	检验镜像的 layer 总数，镜像层总数不能超过 127
mergeAndVerifyConfig	将用户指定的 config 参数与镜像 json 文件中的 config 合并并验证
newContainer	创建新的 container 对象
createRootfs	创建属于 container 对象的 rootfs
ToDisk	将 container 对象 json 化之后写入本地磁盘进行持久化
Register	在 Docker Daemon 中注册该新建的 container 对象

12.3.1 LookupImage

创建并且运行一个 Docker 容器，同样需要原材料，这样的原材料是：Docker 镜像以及用户配置的信息 config 对象。而 LookupImage 函数的功能即为：通过用户指定的镜像名

称，从 daemon 的 repositories 对象中查找镜像。Docker 镜像的概念已经在 Docker Daemon 的多个概念中出现过，如对象 daemon.repositories 的类型为 TagStore，以及类型 Graph 与类型 Driver 等。这三种结构体之间的关系如图 12-2 所示：

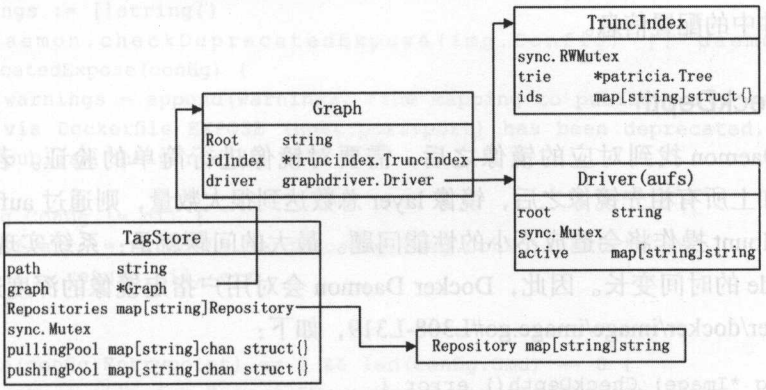


图 12-2　TagStore、Graph 以及 Driver 之间的关系

Docker Daemon 每次成功下载一个含有 tag 的镜像，或者每次成功构建一个镜像，最后的步骤都是将这个镜像在 daemon.Repositories 中注册，若 graghdriver 的类型为 aufs，则镜像注册会写入本地文件 /var/lib/docker/repositories-aufs 文件中。文件 repositories 的存在，使得 LookupImage 函数的实现变得异常简单，LookupImage 函数的源码实现位于 ./docker/docker/graph/tags.go#L79-L97，如下：

```
func (store *TagStore) LookupImage(name string) (*image.Image, error) {
    // FIXME: standardize on returning nil when the image doesn't exist, and
    err for everything else
    // (so we can pass all errors here)
    repos, tag := parsers.ParseRepositoryTag(name)
    if tag == "" {
        tag = DEFAULTTAG
    }
    img, err := store.GetImage(repos, tag)
    store.Lock()
    defer store.Unlock()
    if err != nil {
        return nil, err
    } else if img == nil {
        if img, err = store.graph.Get(name); err != nil {
            return nil, err
        }
    }
    return img, nil
}
```

Docker Daemon 需要用户指定镜像名称，而 LookupImage 函数第一步即解析用户指定镜像的镜像名称与镜像 tag，若 tag 为空，则默认设为“ latest”。紧接着，Docker Daemon 通过 repos（镜像名称）与 tag 信息，尝试从 TagStore 的 graph 属性中提取 img 对象。Docker Daemon 完全通过以上步骤得到 img 对象，从而获取在本地文件系统中的镜像 layer，以及该镜像 json 文件中的配置信息。

12.3.2 CheckDepth

Docker Daemon 找到对应的镜像之后，需要对镜像进行简单的验证。若用户指定的 Docker 镜像加上所有祖先镜像之后，镜像 layer 总数达到很大数量，则通过 aufs 对所有 layer 执行 Union Mount 操作将会造成不小的性能问题。最大的问题就是：系统实现文件读写时，查找文件 inode 的时间变长。因此，Docker Daemon 会对用户指定镜像的深度进行检验，源码位于 ./docker/docker/image/image.go#L308-L319，如下：

```
func (img *Image) CheckDepth() error {
    // We add 2 layers to the depth because the container's rw and
    // init layer add to the restriction
    depth, err := img.Depth()
    if err != nil {
        return err
    }
    if depth+2 >= MaxImageDepth {
        return fmt.Errorf("Cannot create container with more than %d
                           parents", MaxImageDepth)
    }
    return nil
}
```

Docker Daemon 启动容器前，需要通过镜像为容器准备 rootfs，由于通过 aufs 联合 layer 时，会在镜像的 top layer 之上，再叠加两个 layer，分别为 init layer 和容器的 read-write layer。故 CheckDepth 函数计算出指定镜像的深度之后，确保容器镜像的深度值加 2 仍小于最大镜像深度 MaxImageDepth。MaxImageDepth 的值为常量 127。

12.3.3 mergeAndVerifyConfig

分析 Docker 的镜像技术时，有一点内容经常容易被忽略，那就是：Docker 镜像不仅包含文件系统中静态的文件内容，还包含该镜像的 json 文件信息。而 json 文件包含镜像的众多描述信息，包括镜像的 ID、镜像提交时的容器 ID，以及镜像详尽的 Config 配置信息。

Docker 镜像含有 config 信息，docker run 命令传入的许多参数信息也是以 runconfig.Config 的形式转交至 Docker Daemon。

两份 Config 信息如何在运行容器时各自起到相应的作用，是 Docker Daemon 应该考虑的问题。而 mergeAndVerifyConfig 函数则巧妙地将两者有机结合在一起，实现函数为 runconfig

包中的 Merge 函数。函数 mergeAndVerifyConfig 的定义位于 ./docker/docker/dameon/daemon.go#L399-L413，如下：

```
func (daemon *Daemon) mergeAndVerifyConfig(config *runconfig.Config, img *image.
Image) ([]string, error) {
    warnings := []string{}
    if daemon.checkDeprecatedExpose(img.Config) || daemon.check-
    DeprecatedExpose(config) {
        warnings = append(warnings, "The mapping to public ports on your host
        via Dockerfile EXPOSE (host:port:port) has been deprecated. Use -p to
        publish the ports.")
    }
    if img.Config != nil {
        if err := runconfig.Merge(config, img.Config); err != nil {
            return nil, err
        }
    }
    if len(config.Entrypoint) == 0 && len(config.Cmd) == 0 {
        return nil, fmt.Errorf("No command specified")
    }
    return warnings, nil
}
```

函数 mergeAndVerifyConfig 除了 merge 之外，自然还包括 Verify 和 Config，若合并之后的 config 对象中不存在 Entrypoint 并且也没有 Cmd，则说明整个容器没有启动入口，Docker Daemon 必须对这种情况返回错误。

12.3.4　newContainer

Docker Daemon 创建容器对象的第 4 个步骤是：使用 newContainer 函数新建并初始化 container 对象。newContainer 函数的实现过程很清晰，创建并初始化 newContainer 函数内部的 container 对象的源码位于 ./docker/docker/daemon/daemon.go#L529-L548，如下：

```
container := &Container{
    // FIXME: we should generate the ID here instead of receiving it as an argument
    ID:              id,
    Created:         time.Now().UTC(),
    Path:            entrypoint,
    Args:            args, //FIXME: de-duplicate from config
    Config:          config,
    hostConfig:      &runconfig.HostConfig{},
    Image:           img.ID, // Always use the resolved image id
    NetworkSettings: &NetworkSettings{},
    Name:            name,
    Driver:          daemon.driver.String(),
    ExecDriver:      daemon.execDriver.Name(),
    State:           NewState(),
}
```

```
container.root = daemon.containerRoot(container.ID)

if container.ProcessLabel, container.MountLabel, err = label.GenLabels(""); err
!= nil {
    return nil, err
}
```

其中需要特别说明的是 Path 属性，Path 属性的值为 entrypoint。众所周知，entrypoint 是 Docker 容器运行时非常重要的概念。用户一旦指定 entrypoint，则 Docker 运行容器时，可以首先执行 entrypoint 的内容（一般为脚本），而 entrypoint 的最后一行脚本命令一般是 exec 用户指定的 Cmd，届时才开始运行用户指定的应用进程。这样的好处很明显，启动容器时的工作分为两部分：第一，运行 entrypoint，完成与系统配置或初始化相关的工作；第二，运行 Cmd，开始执行用户应用程序。

12.3.5 createRootfs

完成以上四个步骤，原则上已经成功创建了 container 对象。此时 Docker Daemon 并未直接返回 container 对象，而是在此基础上完成容器文件系统 rootfs 的配置。文件系统 rootfs 的挂载，其实要比第 8 章涵盖的内容更多。通过用户指定的镜像，Docker Daemon 完全有能力将所有 layer 通过 aufs 联合挂载起来，而 createRootfs 的实现则是在联合挂载所有镜像 layer 的基础上，再挂载两个 layer，一层称之为“init layer”，另一层则为大家熟知的“read-write layer”。

在 createRootfs 函数中，创建 init layer 以及 read-write layer 的源码实现位于 ./docker/docker/daemon/daemon.go#L568-L574，如下：

```
if err := graph.SetupInitLayer(initPath); err != nil {
    return err
}

if err := daemon.driver.Create(container.ID, initID); err != nil {
    return err
}
```

graph 包的 SetupInitLayer 的作用是：在镜像基础上挂载一系列与镜像无关而与容器运行环境相关的目录和文件，如 /dev/pts、proc、.dockerinit、/etc/hosts、etc/hostname 以及 /etc/resolv.conf 等。需要特殊说明的是：.dockerinit 为 dockerinit 二进制文件挂载点，而 dockerinit 是 Docker 容器中第一个运行的内容。第 13 章将详细分析 dockerinit 在 Docker 容器中的作用。

12.3.6 ToDisk

Docker Daemon 对于每一个创建的容器，均会将其 json 化后持久化至本地文件系统

中。ToDisk 函数的功能正是如此。ToDisk 依靠 toDisk 函数来完成，后者的源代码定义位于 ./docker/docker/daemon/container.go#L112-L129，如下：

```
func (container *Container) toDisk() error {
    data, err := json.Marshal(container)
    if err != nil {
        return err
    }
    pth, err := container.jsonPath()
    if err != nil {
        return err
    }

    err = ioutil.WriteFile(pth, data, 0666)
    if err != nil {
        return err
    }
    return container.WriteHostConfig()
}
```

以上代码清晰明了，它不仅实现 container 对象 json 化后的本地持久化，还实现 container 中 hostConfig 对象的本地持久化，最为重要的自然是确定写入的路径。代码 `pth, err := container.jsonPath()` 即获取 json 文件的放置目录 /var/lib/docker/container/containers/<container_id>。

12.3.7 Register

container 对象创建之后，并根据一些规则对其进行预处理之后，Docker Daemon 将 container 对象注册至 daemon 的 containers 属性。注册的内容为 container 对象的容器 ID。注册的意义在于，Docker Daemon 此后可以通过 daemon 对象调度并管理这个有效的 container 对象。

Register 函数执行完毕，意味着 Docker Daemon 创建容器对象的工作全部完成。如此一来，万事俱备，只欠东风，Docker Daemon 的容器启动命令一触即发。

12.4 Docker Daemon 启动容器

Docker 的世界中，动态的内容往往更加迷人。静态的内容，诸如镜像、Dockerfile、Docker Registry 等，似乎都是为了服务于动态的容器。12.3 节中的 container 对象的创建，也仅仅是静态内容的梳理与组织，仍缺少容器方面的活力。

Docker Daemon 启动容器的操作取决于 docker run 命令发起的第一个 POST 请求。由于 container 对象已经创建，并由 daemon 对象管理，故 POST 请求只要携带 container 的 ID 信息，容器启动命令就能触发。

从逻辑的角度而言，Docker Daemon 仅仅通过获取 container 对象，并执行此对象的 Start 函数。从 Start 函数的具体执行内容而言，涉及的内容很广泛，图 12-3 详细说明 Docker Daemon 针对 container 对象所操作的具体内容。

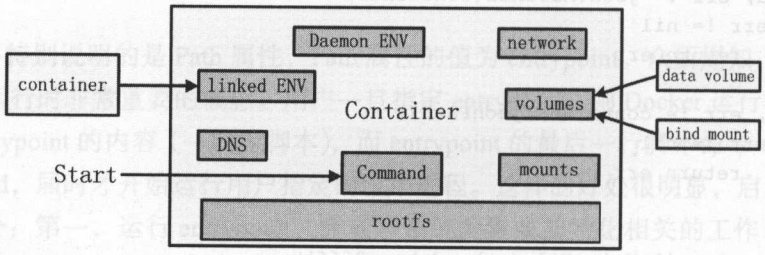


图 12-3　Start 函数涉及内容示意图

就顺序而言，Docker Daemon 执行 Start 函数，包含以下 11 个步骤。

12.4.1　setupContainerDns

一个功能完善的 Docker 容器，肯定离不开基本的网络能力。出色的网络能力意味着容器内部不是一个闭塞的环境，容器可以与外界建立通信。这样的网络自然离不开 DNS 服务，用于可以确保域名的正常工作。Docker Daemon 启动容器的第一个步骤就是配置容器的 DNS 服务。

一旦 Docker 容器的网络模式为 host 模式，则容器和宿主机共享同一个网络命名空间，容器无须为自身的 DNS 服务考虑，完全使用宿主机的 DNS 服务。如若不然，Docker Daemon 则需要为容器内部配置 DNS 服务。配置容器 DNS 的源码实现位于 ./docker/daemon/container.go#L842-L979，如下：

```
func (container *Container) setupContainerDns() error {

    resolvConf, err := resolvconf.Get()
    ...
    container.ResolvConfPath, err = container.getRootResourcePath("resolv.conf")
    ......
    if config.NetworkMode != "host" && (len(config.Dns) > 0 || len(daemon.config.
        Dns) > 0 || len(config.DnsSearch) > 0 || len(daemon.config.DnsSearch) > 0) {
        var (
            dns       = resolvconf.GetNameservers(resolvConf)
            dnsSearch = resolvconf.GetSearchDomains(resolvConf)
        )
        if len(config.Dns) > 0 {
            dns = config.Dns
        } else if len(daemon.config.Dns) > 0 {
            dns = daemon.config.Dns
        }
```

```
        ...
        return resolvconf.Build(container.ResolvConfPath, dns, dnsSearch)
    }
    return ioutil.WriteFile(container.ResolvConfPath, resolvConf, 0644)
}
```

创建容器的 DNS 服务与宿主机有着密不可分的关系。Docker Daemon 首先通过 resolvconf.Get() 获取宿主机在 /etc/resolv.conf 文件中的 DNS 信息；随后在 /var/lib/docker/containers/<container_id>/ 目录下获取 resolv.conf 文件的路径（此文件最终会被挂载至容器内部）；若容器的网络模式不为 host 模式，则紧接着的工作是 Docker Daemon 判断用户和 Docker Daemon 是否指定 DNS 地址，一旦指定，则使用指定的 DNS 地址。

正如上面提及的，Docker 用户启动 Docker Daemon 和启动 Docker 容器时都可以指定 DNS 地址。当用户使用 docker run 命令启动容器时，若使用 -dns 参数，则表明用户需要为容器指定 DNS 地址（在 Docker Daemon 中对应的数据结构为对象 config.Dns），Docker Daemon 优先满足 Docker 用户指定 DNS 的需求。若用户没有指定 -dns 参数，则 Docker Daemon 会为容器配置 Docker Daemon 的 DNS 地址。如果用户在启动 Docker Daemon 时指定 -dns 参数（在 Docker Daemon 中对应的数据结构为 daemon.config.Dns），则表明默认情况下，Docker Daemon 会为 Docker 容器配置该 DNS 地址。当用户启动 Docker Daemon 时，若没有指定 -dns 参数，并且宿主机 /etc/resolv.conf 文件中第一条记录的 DNS 地址为 127.0.0.1 或者 127.0.1.1（容器内部使用这两个 DNS 地址无效），则 Docker Daemon 采用默认的 DNS 地址 8.8.8.8 和 8.8.4.4。采用默认 DNS 地址并不会一劳永逸。特殊情况下，8.8.8.8 和 8.8.4.4 并不能提供稳定可靠的域名解析服务，很大程度上影响 Docker 容器的使用。若用户没有指定任何 -dns 参数，且宿主机的 resolv.conf 中第一条 DNS 记录地址不为 127.0.0.1 和 127.0.1.1，则 Docker Daemon 会为容器配置宿主机的 DNS 地址。

Docker 容器 DNS 地址的问题，不仅仅存在于 docker run 命令中，Dockerfile 的 build 过程中只要涉及 RUN 命令，就也有可能牵扯到 DNS 问题，故不论是启动 Docker Daemon 时的 -dns 参数还是启动 Docker 容器时的 -dns 参数，均不可小觑。

12.4.2　Mount

容器的运行离不开文件系统的支持，而找到容器的根目录则被视为一项基本工作。Docker 环境中，容器的文件系统概念要比平常复杂一些。文件系统在容器看来是一个树型结构，但是对于宿主机乃至 Docker Daemon 而言却并非如此。

Mount() 函数即实现获取容器的根目录，并赋予 container.basefs，container.Mount() 的实现通过 daemon 的 Mount 函数来完成，源码实现位于 ./docker/docker/daemon/daemon.go#L907-L919，如下：

```
func (daemon *Daemon) Mount(container *Container) error {
    dir, err := daemon.driver.Get(container.ID, container.GetMountLabel())
```

```
    if err != nil {
        return fmt.Errorf("Error getting container %s from driver %s: %s",
        container.ID, daemon.driver, err)
    }
    if container.basefs == "" {
        container.basefs = dir
    } else if container.basefs != dir {
        return fmt.Errorf("Error: driver %s is returning inconsistent paths
        for container %s ('%s' then '%s')",daemon.driver, container.ID,
        container.basefs, dir)
    }
    return nil
}
```

12.4.3 initializeNetworking

函数 intializeNetworking 的作用与实现在 7.4.2 节中分析过。总而言之，Docker Daemon 需要处理用户指定的网络模式（bridge、host、other container 和 none 模式中的一种），进行相应的网络配置与初始化，最终将所有初始化后的信息都存入 container 对象中。

12.4.4 verifyDaemonSetting

网络方面的配置，以及诸如容器内存、CPU 限制等参数，都准备完毕之后，Docker Daemon 对 container 对象的 Config 属性进行了进一步的验证，查看是否参数值与 Docker Daemon 的运行环境出现不一致的情况。函数 verifyDaemonSetting 就负责这样的工作，源码实现位于 ./docker/docker/daemon/container.go#L936-L948，如下：

```
func (container *Container) verifyDaemonSettings() {
    if container.Config.Memory > 0 && !container.daemon.sysInfo.MemoryLimit {
        log.Infof("WARNING: Your kernel does not support memory limit
        capabilities. Limitation discarded.")
        container.Config.Memory = 0
    }
    if container.Config.Memory > 0 && !container.daemon.sysInfo.SwapLimit {
        log.Infof("WARNING: Your kernel does not support swap limit
        capabilities. Limitation discarded.")
        container.Config.MemorySwap = -1
    }
    if container.daemon.sysInfo.IPv4ForwardingDisabled {
        log.Infof("WARNING: IPv4 forwarding is disabled. Networking will not work")
    }
}
```

从代码的角度可见，验证的维度主要有三个：系统内核是否支持 cgroup 内存限制，系统内核是否支持 cgroup 的 swap 内存限制，以及系统内核是否支持网络接口间 IPv4 数据包的转发。首先需要明确的是：如何判断系统内核在这三个维度上是否支持。对于 cgroup 内

存限制和 swap 内存限制，Docker Daemon 在启动时即会前往 cgroup 文件系统的挂载点扫描，一旦发现 memory.limit_in_bytes 文件和 memory.soft_limit_in_bytes 文件，则说明系统内核支持 cgroup 内存限制；同理，一旦发现 memory.memsw.limit_in_bytes，则说明系统内核支持 cgroup 的 swap 内存限制。原则上，Docker Daemon 只要找到文件 /proc/sys/net/ipv4/ip_forward，就可以认为系统内核支持 IPv4 包的转发功能，然而在 Docker 1.2.0 版本上却并未很好地完善这一点。

12.4.5　prepareVolumesForContainer

volume 是 Docker 中有关存储的重要概念。了解完 Docker 的镜像原理之后，也许很多人都会认为 Docker 容器中所有内容的存储都会在 aufs 联合挂载的文件系统中。当然，这样的观点是不够准确的。Docker 允许用户从容器外部挂载目录至容器内部，这一方面扩展了容器文件系统的范畴，另一方面拓宽了容器与外界共享资源的方式。对于单个容器而言，用户可以为其配置两种类型的 volume，第一种是 bind-mount 类型的 volume，即用户指定容器外部目录挂载到容器内部的指定目录；另一种是 data volume，即用户只指定 volume 在容器内部的目录，关于 volume 在宿主机上的目录，Docker Daemon 接管创建工作。

volume 的存在使得 Docker 容器的存储可以实现持久化。对于数据存储类的镜像，如 MySQL、MongDB 等，均会采用 data volume，用于持久化数据存储。以 MySQL 为例，MySQL 存储引擎会将数据存储在文件系统的 /var/lib/docker 目录下，而此目录又是通过 data volume 的形式挂载进容器的，实际位置在宿主机文件系统中位于 /var/lib/docker/vfs/dir/<some_id>，因此数据将会存入宿主机文件系统的 /var/lib/docker/vfs/dir/<some_id> 中。更为重要的是，Docker Daemon 使用 docker rm 命令删除容器时，如果不指定 -v 参数，则默认不会删除容器的 data volume，即目录 /var/lib/docker/vfs/dir/<some_id>。长此以往，如果不对 data volume 的磁盘空间进行清理，很容易消耗磁盘空间。

函数 prepareVolumesForContainer 的作用为：通过用户指定的 volume 参数以及镜像中的 data volume 参数，为容器准备更为具体的 Volumes 信息。在 Docker Daemon 中，结构体 Volume 的定义如下：

```
type Volume struct {
    HostPath    string
    VolPath     string
    Mode        string
    isBindMount bool
}
```

一个 Volume 实例均会有 4 个基本属性，HostPath 代表 volume 在宿主机上的路径，VolPath 代表 volume 在容器内部的路径，Mode 代表容器对于 Volume 的读写权限，isBindMount 代表 volume 是否为 bind-mount 类型。

若容器的某个 volume 属于 bind-mount 类型，则用户必须显式指定宿主机上的路径

HostPath、容器内部的路径 VolPath，以及想要的权限。如果 volume 为 data volume，则 data volume 的规范中不允许用户指定 volume 在宿主机上的路径，此时 Docker Daemon 会接管 volume 在宿主机上的路径配置。Docker Daemon 创建 data volume 的函数为 createVolumeHostPath，定义位于 ./docker/docker/daemon/volumes.go#L222-L238，如下：

```
func createVolumeHostPath(container *Container) (string, error) {
    volumesDriver := container.daemon.volumes.Driver()

    // Do not pass a container as the parameter for the volume creation.
    // The graph driver using the container's information ( Image ) to
    // create the parent.
    c, err := container.daemon.volumes.Create(nil, "", "", "", "", nil, nil)
    if err != nil {
        return "", err
    }
    hostPath, err := volumesDriver.Get(c.ID, "")
    if err != nil {
        return hostPath, fmt.Errorf("Driver %s failed to get volume rootfs
        %s: %s", volumesDriver, c.ID, err)
    }

    return hostPath, nil
}
```

Docker Daemon 通过类型为 vfs 的 graphdriver 来管理 data volume，由于 container.daemon.volumes 是指向 graph.Graph 的指针，故 Create 函数位于 graph 包中，功能为创建一个镜像，并在 volumes 这个 graph 中注册，最终通过代码 volumesDriver.Get(c.ID, "") 返回 volume 在宿主机上的路径，路径为 /var/lib/docker/vfs/dir/<some_id>/。

函数 prepareVolumesForContainer 完成的是 mount 信息的准备工作，创建所有的 Volume 对象，并存入 container 对象中。

12.4.6 setupLinkedContainers

容器间的 link 操作允许容器通过环境变量的形式发现另一个容器，并在这两个容器间安全传输信息。用户在启动容器时可以通过设置 --link 参数来完成这项工作，如 docker run --link db:aliasubuntu:14.04，作用是启动容器时为容器 db 设置一个别名 alias，并将别名 alias 以及容器 db 的网络信息配置到新启动容器的环境变量中。

函数 setupLinkedContainers 的作用就是为启动容器配置 link 的环境变量。该函数的的执行流程图如图 12-4 所示。

下面对于图 12-4 中各个步骤的作用进行简单描述。

步骤 1，获取指定容器的子容器，实现代码为 `children, err := daemon.Children(container.Name)`。Docker 创建的容器被 daemon 管理时并非与其他容器毫无关系。一旦容器存在 link 信息，Docker Daemon 对于容器的“父子关系”就会充分体现出

来。被链接的容器将会被视为发起链接的容器的子容器。而这样的关系将会记录在 Docker Daemon 管理的 graphdb 中。关于 Docker Daemon 如何并且何时记录这样的关系，实际位于 container.Start() 的 RegisterLinks 中，有兴趣的读者可以继续深入研读 Docker 关于 graphdb 的容器 link 信息记录。因此，步骤 1 会从 graphdb 中获取所需启动容器的所有子容器，目的是为了将所有子容器的网络信息配置进此容器的环境变量。

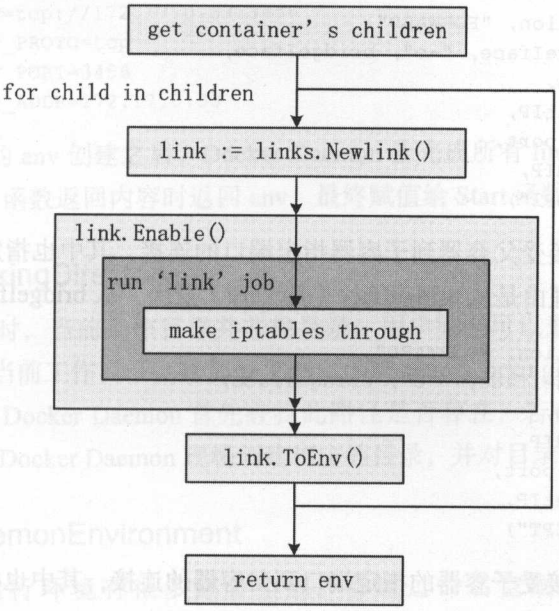


图 12-4　setupLinkedContainers 函数执行流程图

完成步骤 1 之后，Docker Daemon 随即通过 children 中的每一个 child 进行 link 信息 Env 化。本节暂且分析流程中的第一个循环。

步骤 2，创建完备的 link 对象，对象中包含 link 所需要的所有信息，如：父容器的 IP、子容器的 IP、子容器被链接时的别名、子容器 Config 信息中的 Env 属性以及子容器内部暴露的端口号。创建 link 对象的实现源码位于 ./docker/docker/daemon/container.go#L977-L983，如下：

```
link, err := links.NewLink(
    container.NetworkSettings.IPAddress,
child.NetworkSettings.IPAddress,
linkAlias,
child.Config.Env,
child.Config.ExposedPorts,
daemon.eng)
```

步骤 3，使得 link 对象的实际意义生效，开启 link 容器间访问的 iptables 规则。link.Enable() 的实现位于 ./docker/docker/links/link.go，实现手段为创建并执行名为“link”的

job，传入的参数大多为 link 对象的内容。Docker Daemon 启动时，执行网络初始化时，曾注册了 key 为“link”的处理方法，其中 value 为“LinkContainers”。函数 LinkContainer 的内容较为易懂，主要是通过链接两个容器的网络信息，保证即使在 -icc=false 的情况，依旧可以通过开启 iptables 规则，保证容器间指定端口的访问保持畅通。这样的 iptables 规则主要有以下两条：

```
iptables.Raw(action, "FORWARD",
    "-i", bridgeIface, "-o", bridgeIface,
    "-p", proto,
    "-s", parentIP,
    "--dport", port,
    "-d", childIP,
    "-j", "ACCEPT")
```

第一条规则用于接受父容器到子容器指定端口的连接，其中也指定了协议类型，并且表明是容器间的通信，理由是从 bridgeIface（docker0）发出，从 bridgeIface（docker0）接收。

```
iptables.Raw(action, "FORWARD",
    "-i", bridgeIface, "-o", bridgeIface,
    "-p", proto,
    "-s", childIP,
    "--sport", port,
    "-d", parentIP,
    "-j", "ACCEPT")
```

第二条规则用于接受子容器的指定端口到父容器的连接，其中也指定了协议类型，并且表明是容器间的通信，原因与前面一致。

步骤 4，将 link 对象 Env 化，只有最终的 Env 信息，才能被容器内部的进程使用。实现源码为 link.ToEnv()，ToEnv 函数的定义位于 ./docker/docker/links/links.go#L50，而 link 对象 Env 化的主要源码如下：

```
if p := l.getDefaultPort(); p != nil {
    env = append(env, fmt.Sprintf("%s_PORT=%s://%s:%s", alias, p.Proto(),
        l.ChildIP, p.Port()))
}
// Load exposed ports into the environment
for _, p := range l.Ports {
    env = append(env, fmt.Sprintf("%s_PORT_%s_%s=%s://%s:%s", alias,
        p.Port(), strings.ToUpper(p.Proto()), p.Proto(), l.ChildIP,
        p.Port()))
    env = append(env, fmt.Sprintf("%s_PORT_%s_%s_ADDR=%s", alias,
    p.Port(), strings.ToUpper(p.Proto()), l.ChildIP))
    env = append(env, fmt.Sprintf("%s_PORT_%s_%s_PORT=%s", alias,
        p.Port(), strings.ToUpper(p.Proto()), p.Port()))
    env = append(env, fmt.Sprintf("%s_PORT_%s_%s_PROTO=%s", alias,
        p.Port(), strings.ToUpper(p.Proto()), p.Proto()))
}
```

```
    // Load the linked container's name into the environment
    env = append(env, fmt.Sprintf("%s_NAME=%s", alias, l.Name))
```

以上源码也就是在父容器中出现诸多以下环境变量的原因。父容器中关于子容器 link 信息的样例环境变量如下：

```
DB_NAME=/web/db
DB_PORT=tcp://172.17.0.33:3456
DB_PORT_3456_TCP=tcp://172.17.0.33:3456
DB_PORT_3456_TCP_PROTO=tcp
DB_PORT_3456_TCP_PORT=3456
DB_PORT_3456_TCP_ADDR=172.17.0.33
```

众多如以上形式的 env 创建之后，Docker Daemon 会完成所有 link 环境变量的拼接，在 setupLinkedContainers 函数返回内容时返回 env，最终赋值给 Start 函数中的 linkedEnv。

12.4.7 setupWorkingDirectory

Docker 容器运行时，当前工作目录并非根目录，用户完全可以通过传入 --workdir 参数来设置容器运行时的当前工作目录，这部分工作由函数 setupWorkingDirectory 来完成。对于用户指定的 workdir，Docker Daemon 首先查找此路径是否存在，若存在且是目录，则表明 workdir 有效；否则，Docker Daemon 现场创建相应的目录，并对目录添加文件权限。

12.4.8 createDaemonEnvironment

Docker 容器的运行环境有很多部分组成，有通过命名空间实现的隔离环境，有通过控制组实现的资源空间环境，也有用户配置环境变量的运行环境等。函数 createDaemonEnvironment 则为 Docker 容器的运行创建相应的环境变量，这样的环境变量不仅包括系统为容器添加的环境变量，还包括用户为容器指定或生成的环境变量。

函数 createDaemonEnvironment 的源码实现位于 ./docker/docker/daemon/container.go#L1004-L1024，如下：

```
func (container *Container) createDaemonEnvironment(linkedEnv []string) []string {
    // Setup environment
    env := []string{
        "PATH=" + DefaultPathEnv,
        "HOSTNAME=" + container.Config.Hostname,
        // Note: we don't set HOME here because it'll get autoset intelligently
        // based on the value of USER inside dockerinit, but only if it isn't
        // set already (ie, that can be overridden by setting HOME via -e or ENV
        // in a Dockerfile).
    }
    if container.Config.Tty {
        env = append(env, "TERM=xterm")
    }
```

```
        env = append(env, linkedEnv...)
        // because the env on the container can override certain default values
        // we need to replace the 'env' keys where they match and append anything
        // else.
        env = utils.ReplaceOrAppendEnvValues(env, container.Config.Env)

        return env
    }
```

从清晰的代码中，我们可以辨别出 env 对象为一个 string 型数组，数组中的每一项均为 Docker 容器的环境变量。代码表明，这样的环境变量有 PATH、HOSTNAME、TERM 以及用户指定容器 link 之后产生的环境变量。所有的环境变量会在运行容器进程时，加载到进程的环境变量中。

12.4.9 populateCommand

容器需要运行，自然离不开容器运行的入口，即通过运行程序达到容器运行的目的。为此，Docker Daemon 以 Command 的形式提供容器的运行入口，并通过函数 populateCommand 填充 Command 的内容。

第 7 章对 populateCommand 的实现有详尽的介绍，其中较为重要的依然是 Command 的类型以及与其他数据结构的关系。

Docker Daemon 启动容器的多个阶段都对 container 对象的 Config 属性进行配置和处理，而最终这些变化都会体现在 Command 对象实例中，如 initializeNetworking 完成的网络配置将修改 Command 对象中的网络部分，prepareVolumesForContainer 完成的 volume 配置也会在 setupMountsForContainer 函数执行过程中配置进 Command（volume 转换为 mount 对象），setupWorkingDirectory 则设置 Command 对象中的容器进程工作目录等。

容器运行的入口已经配置完毕，Docker Daemon 启动容器也将一触即发。

12.4.10 setupMountsForContainer

函数 prepareVolumesForContainer 的使命是：将用户在 volume 接口配置的 volume 转换为 Docker Daemon 能识别的 volume 类型；而 setupMountsForContainer 的作用是：将 Docker Daemon 中所有需要从容器外挂载到容器内的目录，转换为 execdriver 可以识别的 Mount 类型。函数 setupMountsForContainer 的源码实现位于 ./docker/docker/daemon/volume.go#L51-L75，如下：

```
func setupMountsForContainer(container *Container) error {
    mounts := []execdriver.Mount{
        {container.ResolvConfPath, "/etc/resolv.conf", true, true},
    }
    if container.HostnamePath != "" {
        mounts = append(mounts, execdriver.Mount{container.HostnamePath,
```

```
        "/etc/hostname", true, true})
    }
    if container.HostsPath != "" {
        mounts = append(mounts, execdriver.Mount{container.HostsPath, "/etc/
        hosts", true, true})
    }

    // Mount user specified volumes
    // Note, these are not private because you may want propagation of (un)
    mounts from host
    // volumes. For instance if you use -v /usr:/usr and the host later mounts
    /usr/share you
    // want this new mount in the container
    for r, v := range container.Volumes {
        mounts = append(mounts, execdriver.Mount{v, r, container.
        VolumesRW[r], false})
    }
    container.command.Mounts = mounts
    return nil
}
```

以上代码中，mounts 为 execdriver.Mount 类型的数组，数组的内容均表示容器内部的路径与宿主机路径的挂载映射关系，并标明读写权限与私有权限。容器内部的挂载点一般包括 /etc/resolv.conf、/etc/hostname、/etc/hosts 以及所有的 volume 等。最终，mounts 对象赋予 container.command 的 Mounts 属性，为 Command 类型实例服务。

12.4.11　waitForStart

Docker Daemon 为启动容器所做的准备工作不可谓不充分。准备工作全部完成之时，也是启动容器、运行容器之时。函数 waitForStart 的功能就是启动容器进程，并根据用户指定的重启策略应对容器启动失败的情况。

函数 waitForStart 的定义位于 ./docker/docker/daemon/container.go#L1070-L1082，如下：

```
func (container *Container) waitForStart() error {
    container.monitor = newContainerMonitor(container, container.hostConfig.
    RestartPolicy)
    // block until we either receive an error from the initial start of the
    container's
    // process or until the process is running in the container
    select {
    case <-container.monitor.startSignal:
    case err := <-utils.Go(container.monitor.Start):
        return err
    }
    return nil
}
```

Docker Daemon 为 container 创建了一个 containerMonitor 类型实例，并将其赋予 container 对象的 monitor 属性。类型 containerMonitor 的定义位于 ./docker/docker/daemon/monitor.go，在启动容器时起到监控容器主进程的角色。如果 Docker 用户为容器的启动配置了重启策略，则 containerMonitor 会确保主进程启动失败时基于重启策略而重启。然而，若容器由于某种原因在重启策略下依旧重启失败，则 containerMonitor 将重置以及清理 container 对象占据的资源，如为容器分配的网络资源、为容器创建的 rootfs 等。

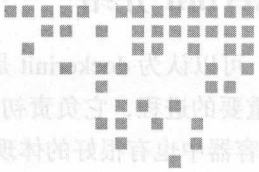
理解 containerMonitor 的监控作用之后，还需要关注其启动容器的本质，源代码为 container.monitor.Start。Start 函数的作用很明显，即通过启动 populateCommand 函数创建的 Command 对象，完成容器的创建。原理如此，而实现却要更复杂一些，代码的执行流首先会进入 execdriver(native)，execdriver 为了适配 libcontainer 的接口，需要根据 container 中的 Command 对象创建 libcontainer 的 Config 对象，最终通过 libcontainer 中的 namespaces 包来实现容器的启动。

创建容器对象，仅仅是为 Docker 容器组织了必需的配置信息，并未将配置信息落实至 Docker 容器运行环境中的具体资源与能力。因此，libcontainer 在启动容器时，一方面为容器初始化具体的物理资源并提供相应的容器能力，另一方面启动容器的主进程。第 13 章将分析 Docker 容器主进程 dockerinit 以及其他相关内容。

12.5 总结

Docker 容器的启动，意味着用户运行时的开始。对于 Docker 用户而言，仅仅能感受到自身应用的正常运行。然而，对于 Docker Daemon 而言，需要完成的工作要远比这复杂，从最初的镜像查找，到 rootfs 配备，还有网络、volume、容器 link 等配置信息的收集，以及最后启动代表容器主进程的 Command 命令。

Docker 体系内的万千概念，最后都融汇在代表容器主进程的 Command 命令中，且最终体现在主进程的启动流程中。本章从 Docker 容器的创建入手，着重分析了 Docker Daemon 的启动全过程。



Chapter 13

第 13 章 dockerinit 启动

13.1 引言

Docker 容器作为 Docker Daemon 管理的对象，给用户的体验是一个隔离性较好的运行环境。作为容器技术，隔离环境背后的技术支撑是什么？ Docker 容器可以运行用户应用进程，容器与进程的关系又如何？可以说，认识 Docker 是一条漫长的路，在这条路上，仍然存在不少疑惑。

为阐述 Docker 容器与进程的关系，首先以用户都见过的 docker run 命令为例。通过 Docker 容器完成指定应用进程的运行，相信所有的 Docker 用户都不陌生，然而，应用进程是否是容器内的第一个进程，一旦提及这一点，也会激起 Docker 用户心中的疑惑。虽然第 7 章已经对容器与进程的关系进行了初步的介绍与分析，但是关于容器内第一个进程究竟为何方神圣，依然没有揭开神秘的面纱。开门见山，Docker Daemon 启动 Docker 容器时，运行的第一个进程并不是用户的应用进程，而是一个称为 dockerinit 的进程。

本章主要从源码的角度介绍 dockerinit，并分析 dockerinit 启动流程的实现。本书基于 Docker 1.2.0 版本，libcontainer 的版本为 1.2.0。本章内容主要包含以下 4 个方面：

1）简要介绍 dockerinit；

2）分析 dockerinit 的执行入口；

3）分析 dockerinit 的运行；

4）分析 libcontainer 的运行。

13.2 dockerinit 介绍

正如其名，可以认为 dockerinit 是 Docker 容器中的 init 进程。Linux 操作系统中，init 进程是一个非常重要的进程，它负责初始化系统，并且是所有用户进程的祖先进程。相同的概念，在 Docker 容器中也有很好的体现，dockerinit 作为 Docker 容器中的第一个进程，同样扮演初始化容器的角色，同样是 Docker 容器内所有进程的祖先进程。

13.2.1 dockerinit 初始化内容

既然 dockerinit 扮演初始化容器的角色，那么具体初始化容器的哪些内容必将受到关注。对于 Docker 容器而言，容器的资源以及容器的能力是重点，而分配的资源以及授予的能力，都应该在容器创建时初始化。以下是 dockerinit 初始化的部分资源与能力：

- 网络资源。Docker Daemon 为 bridge（桥接）模式下的容器创建网络命名空间，创建初期命名空间内并无网络栈（如网络接口等），dockerinit 需要为容器初始化网络命名空间，配置独立的网络栈；
- 挂载资源。容器都有独立的 mount namespace，初始化挂载资源包括设置容器内部的设备、挂载点以及文件系统等；
- 用户设置。容器都属于一个用户，dockerinit 负责为容器设置组（group）、组 ID（GID）与用户 ID（UID）；
- 环境变量。容器中的环境变量使得容器内进程拥有更多的运行参数。
- 容器 Capability。容器中用户使用的内核与宿主机无差别，Linux 的 Capability 机制则可以确保容器内进程以及文件的 Capability 得到限制。

13.2.2 dockerinit 与 Docker Daemon

dockerinit 是 Docker 容器中的第一个进程。Linux 操作系统中，进程的诞生往往都由 fork 系统调用产生，dockerinit 进程自然也不会例外。如此说来，dockerinit 的父进程最有可能是 Docker Daemon，现实情况的确如此。

父子关系是 Docker Daemon 与 dockerinit 之间最明了的关系。Docker Daemon 完成进程 fork 之后，Docker Daemon 是父进程，dockerinit 是子进程；而 exec 子进程则完成容器的初始化工作。关于容器的初始化工作，需要清楚的是 dockerinit 的初始化依据来源于何处，例如，dockerinit 需要初始化网络栈，在自身网络命名空间中创建独立的网络设备，那网络设备的配置信息从何而来？虽然新建了 mount 命名空间，但是命名空间的挂载点位于何处？进程 dockerinit 仍然需要足够的依据，保证初始化工作的顺利进行。

既然是父子关系，dockerinit 被派生出来时，Docker Daemon 即可以将众多的配置信息通过很多有效的途径传递至 dockerinit。为了使得 dockerinit 得到配置信息，Docker Daemon 将容器所有的 Config 配置先 json 化之后，存入本地文件系统中，而 dockerinit 在初始化 mount

命令空间前，提取这部分信息。另外，Docker Daemon 为 dockerinit 传递命令参数也是一种简单的方式。

13.3　dockerinit 执行入口

第 12 章主要分析 Docker 容器的创建，然而，关于 Docker Daemon 启动容器，分析到创建 monitor 对象并监控容器的启动之后，却并未再深入。若继续深究，我们则可以发现程序的执行将从 Docker Daemon 转移至 execdriver，最终进入 libcontainer。

寻找 dockerinit 的诞生，必须追朔到 execdriver。Docker Daemon 启动容器的参数，最终需要 libcontainer 来加载，而 execdriver 负责将 Docker Daemon 描述容器的 container 对象转义，使其符合 libcontainer 的接口。在这样的过程中，execdriver 调用 libcontainer 的 namespace 包时，将 dockerinit 作为参数传递至 libcontainer。调用函数的定义位于 libcontainer 项目的 namespace 包中，函数名为 exec，源码实现位于 ./docker/libcontainer/namespaces/exec.go#L24，如下：

```
func Exec(container *libcontainer.Config, stdin io.Reader, stdout, stderr
io.Writer, console string, rootfs, dataPath string, args []string, createCommand
CreateCommand, startCallback func()) (int, error)
```

13.3.1　createCommand 分析

函数 exec 传入的众多参数中，与 dockerinit 相关的是形参 createCommand，而 execdriver 调用 namespaces.exec() 函数时，实参如下：

```
func(container *libcontainer.Config, console, rootfs, dataPath, init string,
child *os.File, args []string) *exec.Cmd {
    c.Path = d.initPath
    c.Args = append([]string{
        DriverName,
        "-console", console,
        "-pipe", "3",
        "-root", filepath.Join(d.root, c.ID),
        "--",
    }, args...)

    // set this to nil so that when we set the clone flags anything else is reset
    c.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
    }
    c.ExtraFiles = []*os.File{child}

    c.Env = container.Env
    c.Dir = c.Rootfs
```

```
        return &c.Cmd
}
```

可见，传入的实参为一个函数，此函数的作用是：使得 libcontainer 执行 namespaces.exec() 时通过该函数创建一个 exec.Cmd 对象，以便 libcontainer 直接通过 Start 函数，启动该 exec.Cmd 对象。下面分析 exec.Cmd 类型实例 c 的各属性。

.c.Path。对于类型 exec.Cmd 而言，唯一不能为空的就是 Path 参数，它代表执行命令所在路径，而 d.initPath 的值即为 Path 路径。若 Docker 的版本为 1.2.0，execdriver 的类型为 native，那么默认情况下，d.initPath 的值为 /var/lib/docker/init/dockerinit-1.2.0。因此，Docker Daemon 启动 Docker 容器时，均会执行该路径下的 dockerinit-1.2.0。

.c.Args。执行命令的参数同样重要，.c.Args 即代表 dockerinit-1.2.0 命令的执行参数。其中 .c.Args 是一个 string 数组，内容如表 13-1 所示。

表 13-1 .c.Args 参数的内容

参数	参数含义
DriverName	代表 execdriver 的具体类型，默认情况下值为“native”，此参数会在 dockerinit 执行 reexec.Init() 时使用到
- console	名为“console”的 flag，值为紧接其后的元素
console	console（pty slave）的路径
-pipe	名为“pipe”的 flag，代表同步管道的文件描述符 fd
3	名为“pipe”的 flag 的值，代表同步管道的 fd 值为 3
-root	名为“root”的 flag，代表容器配置文件所在的路径
filepath.Join(d.root, c.ID)	名为“root”的 flag 的值，一般为 /var/lib/docker/execdriver/native/<c.ID>，该路径下存放着文件 container.json 和 state.json，前者存储容器配置对象 Config 的 json 信息；后者存储容器的 init 进程信息、网络信息以及控制组路径等
args	用户指定的命令，包含 ENTRYPOINT 与 CMD，在 dockerinit 执行的后期阶段扮演重要角色，实现系统初始化工作转变为用户进程的运行

.c.SysProcAttr。SysProcAttr 将携带 exec.Cmd 运行的操作系统参数，如在启动 dockerinit 时，代表是否需要为进程创建新 namespace 的参数 Cloneflags。关于 namespace 的 Cloneflags 共有 6 个，分别是 syscall.CLONE_NEWNS、syscall.CLONE_NEWUTS、syscall.CLONE_NEWIPC、syscall.CLONE_NEWUSER、syscall.CLONE_NEWPID 和 syscall.CLONE_NEWNET，而默认情况下 Docker 1.2.0 并未完全支持用户命名空间，故没有使用 syscall.CLONE_NEWUSER。

.c.ExtraFiles。ExtraFiles 将携带从父进程 Docker Daemon 继承的文件描述符，由于不包括标准输入（文件描述符为 0）、标准输出（文件描述符为 1）和标准错误（文件描述符为 2），故 ExtraFile 的值都是不小于 3 的文件描述符。由于在 dockerinit 的运行初期仍然需要与之通信，故 Docker Daemon 创建了一个管道，并将管道的一端交给 dockerinit。

.c.Env。Env 用于指定 dockerinit 进程运行时的环境变量。故用户运行 docker run 命令时

通过 -e 设定的 ENV 参数或者 Dockerfile 中的 ENV 参数，都会在此时用来设定 dockerinit 进程的环境变量。

.c.Dir。Dir 用于指定命令 exec.Cmd 的工作目录，具体的值为 execdriver.Command 的 rootfs。

13.3.2　namespace.exec

有了 createCommand 的基础，理解 libcontainer 中的 namespace.Exec 就变得简单很多。namespace.Exec 的作用无非是通过 createCommand 函数，创建一个容器启动命令 exec.Cmd，随后启动该命令完成 dockerinit 进程的运行工作。容器命令启动的源码实现位于 ./docker/libcontainer/namespaces/exec.go#L24-L47，如下：

```
func Exec(container *libcontainer.Config, stdin io.Reader, stdout, stderr
io.Writer, console string, rootfs, dataPath string, args []string, createCommand
CreateCommand, startCallback func()) (int, error) {
    var (
        err error
    )

    // create a pipe so that we can syncronize with the namespaced process and
    // pass the veth name to the child
    syncPipe, err := syncpipe.NewSyncPipe()
    if err != nil {
        return -1, err
    }
    defer syncPipe.Close()

    command := createCommand(container, console, rootfs, dataPath, os.Args[0],
    syncPipe.Child(), args)
    command.Stdin = stdin
    command.Stdout = stdout
    command.Stderr = stderr

    if err := command.Start(); err != nil {
        return -1, err
    }
    ......
    if err := command.Wait(); err != nil {
        if _, ok := err.(*exec.ExitError); !ok {
            return -1, err
        }
    }
    ......
}
```

以上代码为仅仅是 namespace.Exec 函数执行的前面一部分，代码结构较为清晰，总体分为三个部分：创建 syncPipe、创建容器命令以及容器命令的启动。

顾名思义，创建 syncPipe 的作用是：使得 Docker Daemon 与 dockerinit 可以通过管道的形式同步资源。由于一旦 dockerinit 进程启动，dockerinit 进程的运行就处于多个全新的 namespace 内，此时 Docker Daemon 与 dockerinit 处于相互隔离的 namespace 中，两者的通信变得不便（虽然 dockerinit 处于全新的网络命名空间下，但网络命名空间下网络栈仍未初始化，没有网络接口用于通信），故 Docker 采用更为底层的管道机制完成操作系统上分别处于不同 namespace 进程的通信。syncPipe 主要用于 Docker Daemon 向 dockerinit 传递容器信息，如容器内网络设备的信息等。

创建容器命令即为 createCommand 函数的实现，前面已经涉及，并对最终返回的 exec.Cmd 类型实例进行详细分析。总而言之，createCommand 函数为容器创建了静态的执行入口。

容器命令的启动通过 command.Start() 实现。Start 函数负责启动指定的命令 command，但是不对该命令的运行执行 Wait 操作，Wait 操作有专门的 Wait 函数来完成。由于 command 的 Path 参数为 d.initPath，即 /var/lib/docker/init/dockerinit-1.2.0，故 libcontainer 会启动 dockerinit-1.2.0 命令，换言之，dockerinit 的执行被触发。

分析至此，我们已经找到 dockerinit 的执行入口，但这并不意味着 Docker Daemon 会将所有执行权交给 dockerinit，Docker Daemon 关于容器的启动还远没有结束。启动 dockerinit 之后，Docker Daemon 与 dockerinit 的执行将并发执行，两者之间通过管道的形式进行同步，同步完成之后，Docker Daemon 与 dockerinit 各自运行，除父子关系之外，无更多逻辑关系。13.4 节将深入分析 dockerinit 的执行以及 dockerinit 与 Docker Daemon 的协同通信。

13.4 dockerinit 运行

dockerinit 作为 Docker 容器中运行的第一个进程，本质上，它是一个使用 go 语言编译的可执行文件，即 dockerinit-1.2.0。既然如此，回到 dockerinit 的实现函数，可以发现 dockerinit 的实现全部依靠 reexec.Init() 函数。dockerinit 的 main 函数位于 ./docker/docker/dockerinit/dockernit.go#L9-L12，如下：

```
func main() {
    // Running in init mode
    reexec.Init()
}
```

以上代码简洁得有点极致，main 通过 reexec.Init() 来实现。是否 reexec.Init() 真的实现了对容器环境的初始化？让我们带着疑惑进入 reexec.Init() 的分析。

13.4.1 reexec.Init() 的分析

reexec.Init() 在本书中并非第一次出现，早在第 2 章中，创建 Docker Client 时就涉及了 reexec.Init()，当时在 docker 的 main 函数的运行中，第一个步骤即检验 reexec.Init() 的值，第

二个步骤才是解析 flag 参数，由于还没有 execdriver 等概念的基础，故没有深入。具体源码位于 ./docker/docker/docker.go#L26-L111，如下：

```
func main() {
    if reexec.Init() {
        return
    }
    flag.Parse()
    // FIXME: validate daemon flags here
    ......
}
```

Docker 中包 reexec 的存在，主要是因为使用 Go 在派生进程时存在一些语言层的限制。功能上，reexec 使得 dockerinit 这个二进制文件可以运行多种任务。dockerinit 运行的处理方法都通过一个注册的 execdriver 名称来获取，并最终运行此处理方法。

首先进入 reexec 包中的 Init() 函数定义，源码实现位于 ./docker/docker/reexec/reexec.go#L23-L32，如下：

```
func Init() bool {
    initializer, exists := registeredInitializers[os.Args[0]]
    if exists {
        initializer()
        return true
    }
    return false
}
```

以上代码表明：Init() 将会执行注册的 initializer。既然是注册的 initializer，就会存在一个问题：“Docker 中哪一模块注册了相应的 initializer?”。由于 reexec 包的功能大部分与 exec 相关，而在 Docker 架构中，与 exec 关系最紧密的自然是 execdriver，execdriver 负责执行容器的相应操作，而问题的答案恰巧的确是 execdriver。

以默认的 execdriver 为例，也就是类型为 native 的 execdriver，Docker Daemon 启动时自然会初始化 execdriver，而在初始化类型为 native 的 execdriver 时，会执行 execdriver 对 reexec 的注册，代码位于 ./docker/docker/daemon/execdriver/native/init.go#L19-L21，如下：

```
func init() {
    reexec.Register(DriverName, initializer)
}
```

函数 init() 实现了 DriveName 的注册，DriveName 的值为“native”，而注册时相应的处理方法为 initializer，此 initializer 才是 dockerinit 真正运行的内容。反观 createCommand 环节，dockerinit 的参数中第一个是 DriveName（“native”），故 dockerinit 运行时将直接执行 key 为“native”的注册处理方法。而在 ./docker/docker/docker.go 的 main 函数中，一旦由于某种异常原因，reexec.Init() 已经有注册的值，而 Docker Daemon 启动的过程中仍然需要向

reexec 注册相应的 execdriver，两者很有可能造成冲突。因此，docker.go 中 main 函数确保当前没有注册 execdriver 之后，再继续往下执行。

13.4.2 dockerinit 的执行流程

dockerinit 的执行流程，可以说就是 initializer 的执行流程。在功能方面，initializer 依旧完成一部分初始化工作，并在最后把控制权交给 libcontainer 的 namespace，由后者完成剩余容器的初始化工作。

类型为“native”的 initializer 的源码实现位于 ./docker/docker/daemon/execdriver/native/init.go#L19-L21，如下：

```
func initializer() {
    runtime.LockOSThread()
    var (
        pipe    = flag.Int("pipe", 0, "sync pipe fd")
        console = flag.String("console", "", "console (pty slave) path")
        root    = flag.String("root", ".", "root path for configuration files")
    )
    flag.Parse()

    var container *libcontainer.Config
    f, err := os.Open(filepath.Join(*root, "container.json"))
    if err != nil {
        writeError(err)
    }
    if err := json.NewDecoder(f).Decode(&container); err != nil {
        f.Close()
        writeError(err)
    }
    f.Close()
    rootfs, err := os.Getwd()
    if err != nil {
        writeError(err)
    }
    syncPipe, err := syncpipe.NewSyncPipeFromFd(0, uintptr(*pipe))
    if err != nil {
        writeError(err)
    }
    if err := namespaces.Init(container, rootfs, *console, syncPipe, flag.
    Args()); err != nil {
        writeError(err)
    }
    panic("Unreachable")
}
```

分析 initializer 的实现，发现执行流程包含 5 个步骤，分别如下：

1）定义 flag 参数并解析；

2）声明 libcontainer.Config 实例，并通过解析 container.json 文件，获取实例内容。文件 container.json：execdriver 中 run 函数执行 namespaces.exec 函数之前，将 libcontainer.Config 对象实例持久化至本地目录 /var/lib/docker/execdriver/native/<c.ID>/container.json；

3）获取 root 的路径；

4）通过同步管道所在的文件描述符索引值（值为 3），获取对应的管道对象；

5）通过 libcontainer 中 namespaces 包的 Init 函数，最终完成容器初始化工作。

13.5　libcontainer 的运行

libcontainer 是一套容器技术的实现方案，借助于 libcontainer 的调用，可以完成容器的创建与管理。dockerinit 就通过 libcontainer 来完成容器的创建与初始化。

dockerinit 通过 namespaces 包的 Init 函数来调用 libcontainer 的 API 接口并完成所有工作，但是 Init 完成的内容绝不仅仅只有与 Linux namespace 相关的内容。Init 是全新 namespace 下运行的第一部分内容，同时会完成 mount 信息、容器用户以及网络等多方面的配置。

由于 dockerinit 需要与 Docker Daemon 进行信息同步，故几乎可以将这父子进程的管道同步作为一个分水岭，前半部分作为进程的配置，后半部作为初始化容器的资源。Init 函数的实现位于 ./docker/libcontainer/namespaces/init.go#L32-L122，如下：

```
func Init(container *libcontainer.Config, uncleanRootfs, consolePath string,
syncPipe *syncpipe.SyncPipe, args []string) (err error) {
    ......
    if err := LoadContainerEnvironment(container); err != nil {
        return err
    }

    // We always read this as it is a way to sync with the parent as well
    var networkState *network.NetworkState
    if err := syncPipe.ReadFromParent(&networkState); err != nil {
        return err
    }

    ......
    if err := setupNetwork(container, networkState); err != nil {
        return fmt.Errorf("setup networking %s", err)
    }
    ......

    if err := mount.InitializeMountNamespace(rootfs,
        consolePath,
        container.RestrictSys,
        (*mount.MountConfig)(container.MountConfig)); err != nil {
        return fmt.Errorf("setup mount namespace %s", err)
```

```
    }
    ......
    if err := FinalizeNamespace(container); err != nil {
        return fmt.Errorf("finalize namespace %s", err)
    }

    // FinalizeNamespace can change user/group which clears the parent death
    // signal, so we restore it here.
    if err := RestoreParentDeathSignal(pdeathSignal); err != nil {
        return fmt.Errorf("restore parent death signal %s", err)
    }

    return system.Execv(args[0], args[0:], os.Environ())
}
```

dockerinit 通过向 syncPipe 读取父进程 Docker Daemon 传递的内容，完成两者信息的同步。Docker Daemon 与 dockerinit 的同步流程如图 13-1 所示。

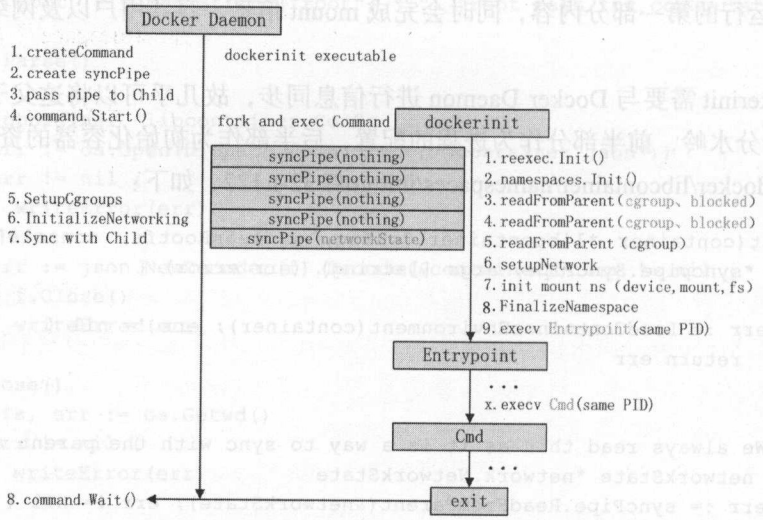


图 13-1　Docker Daemon 与 dockerinit 的同步流程

通过图 13-1 我们可以发现，Docker Daemon 通过 libcontainer 中 namespaces 包的 exec 函数执行 command.Start() 之后，仍有其他的工作需要完成，主要工作如下：

1）SetupCgroups，设置 dockerinit 进程的 cgroups，完成 dockerinit 进程的资源限制；

2）InitializeNetworking，创建 dockerinit 所在容器的网络栈资源；

3）syncPipe.ReadFromChild，与子进程 dockerinit 同步。

回到 Docker Daemon 的 namespaces.Exec 中，下面分析执行流程：

13.5.1　Docker Daemon 设置 cgroups 参数

SetupCgroups 通过类型为 libcontainer.Config 的 container 对象以及 dockerinit 进程在宿主机上的 PID 号，为 dockerinit 设置 cgroups 参数，从此 cgroups 对 dockerinit 进程的资源限制生效，而之后 dockerinit 进程派生出的子进程也将受到相同的资源限制。

资源的使用受到应有的限制，是容器技术的一大亮点。实现原理则是在进程运行之后，通过将进程的 PID 放置于特定的 cgroups 子系统来完成。

13.5.2　Docker Daemon 创建网络栈资源

创建容器网络栈资源依靠函数 InitializeNetworking，调用者 Docker Daemon 一般位于 init 命名空间中，而 dockerinit 位于新建的网络命名空间中，因此 InitializeNetworking 的作用是在 init 命名空间中创建 dockerinit 所在容器需要的网络栈资源，即跨越网络命名空间创建相应的网络栈。原则上，在 go 语言的运行时中，并不具备跨 namespace 的资源操纵能力，故 Docker Daemon 将创建完毕的网络栈资源通过 syncPipe 的形式传递给处于新建 namespace 下的 dockerinit，由 dockerinit 负责接收并初始化新建 namespace 下的网络栈。关于 InitializeNetworking 函数的具体实现，可参考第 7 章。

Docker Daemon 通过管道将网络栈信息传递至 dockerinit 之后，容器启动使命就完成了。Docker Daemon 退居幕后之时，dockerinit 才刚开始大展手脚。

13.5.3　dockerinit 配置网络栈

虽然 Docker Daemon 已经完成自己的使命，但是 dockerinit 依旧处于阻塞状态，而且等待着 SyncPipe 中 Docker Daemon 传来的内容。只要从 syncPipe 中读到内容，dockerinit 即从阻塞状态恢复执行。在设置 SID（Session ID）、处理 tty 等一系列操作之后，dockerinit 最为重要的工作就是配置网络栈，即通过 Docker Daemon 在 syncPipe 中传递来的 NetworkState 信息，配置容器所在网络命名空间下的网络栈。

首先来看 NetworkState 的定义，位于 ./docker/libcontainer/network/types.go#L34-L41，如下：

```
type NetworkState struct {
    // The name of the veth interface on the Host.
    VethHost string `json:"veth_host,omitempty"`
    // The name of the veth interface created inside the container for the child.
    VethChild string `json:"veth_child,omitempty"`
    // Net namespace path.
    NsPath string `json:"ns_path,omitempty"`
}
```

源码注释对于 NetworkState 各属性的描述非常到位，结合 Docker 的网络模式来分析，便可发现：VethHost 和 VethChild 属于 bridge 网络模式下的 veth pair，前者是宿主机上依附

在网桥 docker0 上的 veth，而后者是容器内部的 veth；NsPath 属于 other container 网络模式下其他容器的网络命名空间路径。之所以 NetworkState 中没有 host 网络模式以及 none 网络模式的信息，原因很简单，host 网络模式不需要为容器创建新的网络命名空间，而 none 网络模式不做任何网络配置。

以 bridge 网络模式为例，SetupNetwork 函数正是利用 NetworkState 中的网络接口（veth pair）配置信息，在容器的网络命名空间下初始化内部网络接口，将其改名为 eth0，并对网络接口的 MTU、IP 地址以及默认网关地址进行配置。

SetupNetwork 函数的实现过程中，主要通过 NetworkStrategy 从 container.Networks 对象中的 Type 获取所需初始化的网络接口类型，并通过具体类型进行相应的配置。若网络接口类型为 veth，则网络配置工作的源码实现位于 ./docker/libcontainer/network/veth.go#L52-L77，如下：

```
func (v *Veth) Initialize(config *Network, networkState *NetworkState) error {
    var vethChild = networkState.VethChild
    if vethChild == "" {
        return fmt.Errorf("vethChild is not specified")
    }
    if err := InterfaceDown(vethChild); err != nil {
        return fmt.Errorf("interface down %s %s", vethChild, err)
    }
    if err := ChangeInterfaceName(vethChild, defaultDevice); err != nil {
        return fmt.Errorf("change %s to %s %s", vethChild, defaultDevice, err)
    }
    if err := SetInterfaceIp(defaultDevice, config.Address); err != nil {
        return fmt.Errorf("set %s ip %s", defaultDevice, err)
    }
    if err := SetMtu(defaultDevice, config.Mtu); err != nil {
        return fmt.Errorf("set %s mtu to %d %s", defaultDevice, config.Mtu, err)
    }
    if err := InterfaceUp(defaultDevice); err != nil {
        return fmt.Errorf("%s up %s", defaultDevice, err)
    }
    if config.Gateway != "" {
        if err := SetDefaultGateway(config.Gateway, defaultDevice); err != nil {
            return fmt.Errorf("set gateway to %s on device %s failed with
                %s", config.Gateway, defaultDevice, err)
        }
    }
    return nil
}
```

分析以上 Initialize 函数的实现，很快可以总结出以下执行流程：

1）将名为 VethChild 的网络接口停止运行；

2）将名为 VethChild 的网络接口改名为 defaultDevice，即 eth0；

3）为网络接口 eth0 设置 IP 地址；

4）为网络接口 eth0 设置 MTU 值；

5）启动网络接口 eth0；

6）若需设置网关地址，则为网络接口设置默认网关地址。

通过以上 6 个步骤的初始化与配置，容器网络命名空间内的网络接口即成功运行，为容器内部提供完整的网络栈，并且该 veth 网络接口对其他容器不可见，处于独立的容器环境中。

另外，需要说明的是，对于容器技术而言，只要新建一个网络命名空间，容器内部就会默认创建一个 loopback 网络环回接口，供容器内部本地通信。

13.5.4　dockerinit 初始化 mount namespace

对于 dockerinit 而言，不仅网络命名空间下的网络栈资源需要初始化与配置，其他命名空间下的资源同样需要初始化。函数 InitializeMountNamespace 则用于完成挂载资源的初始化。

Docker 体系中，关于挂载资源，无外乎有三种基本的资源：设备（device）、文件系统以及挂载点（mount point）。文件系统资源比较好理解，Docker 容器的运行离不开文件系统，这样的文件系统自然包括容器最初的 rootfs 文件系统，当然也包含 proc 等虚拟文件系统。除此之外，如果对 Docker 中 volume 概念熟悉，理解挂载点（mount point）自然也不是难事，挂载点保证宿主机上的文件资源同样可以被挂载到容器内部，并被容器内部进程使用。设备文件的初始化，实则是为容器的设备创建一个索引节点并进行关联。

函数 InitializeMountNamespace 的定义位于 ./docker/libcontainer/mount/init.go#L29，如下：

```
func InitializeMountNamespace(rootfs, console string, sysReadonly bool,
mountConfig *MountConfig) error
```

其中 mountConfig 参数同样来自与 container.MountConfig，而 container 的类型为 libcontainer.Config，它正是 execdriver 为了适配 libcontainer 通过将 execdriver.Command 转义后的 libcontainer.Config。因此，用户态所有的 mount 信息，最终都会在 libcontainer 执行 InitializeMountNamespace 时有所体现。

13.5.5　dockerinit 完成 namespace 配置

dockerinit 的历史使命并非仅仅是初始化与配置容器的 namespace，完成 namespace 的相应工作之后，仍然需要执行用户指定的 Entrypoint 以及 Cmd。两项任务的分隔线就以函数 FinalizeNamespace 为准。

顾名思义，在开始执行用户命令之前，函数 FinalizeNamespace 完成容器 namespace 下所需的所有工作。函数定义位于 ./docker/libcontainer/namespaces/init.go#L211-L249，如下：

```
func FinalizeNamespace(container *libcontainer.Config) error {
    // Ensure that all non-standard fds we may have accidentally
    // inherited are marked close-on-exec so they stay out of the
```

```
    // container
    if err := utils.CloseExecFrom(3); err != nil {
        return fmt.Errorf("close open file descriptors %s", err)
    }

    // drop capabilities in bounding set before changing user
    if err := capabilities.DropBoundingSet(container.Capabilities); err != nil {
        return fmt.Errorf("drop bounding set %s", err)
    }

    // preserve existing capabilities while we change users
    if err := system.SetKeepCaps(); err != nil {
        return fmt.Errorf("set keep caps %s", err)
    }

    if err := SetupUser(container.User); err != nil {
        return fmt.Errorf("setup user %s", err)
    }

    if err := system.ClearKeepCaps(); err != nil {
        return fmt.Errorf("clear keep caps %s", err)
    }

    // drop all other capabilities
    if err := capabilities.DropCapabilities(container.Capabilities); err != nil {
        return fmt.Errorf("drop capabilities %s", err)
    }

    if container.WorkingDir != "" {
        if err := syscall.Chdir(container.WorkingDir); err != nil {
            return fmt.Errorf("chdir to %s %s", container.WorkingDir, err)
        }
    }

    return nil
}
```

以上代码逻辑全部顺序执行，逐步分析执行步骤，可以得到以下结果：

1）CloseExecFrom(3)，关闭除标准输入（文件描述符为 0）、标准输出（文件描述符为 1）、标准错误（文件描述符为 2）之外所有打开的文件描述符（文件描述符大于等于 3）；

2）DropBoundingSet，在容器切换用户之前，为容器取消某些 Linux Capability；

3）SetKeepCaps，在容器切换用户前，为容器保留已经拥有的 Linux Capability；

4）SetupUser，为容器创建新的用户 ID、组 ID、补充组的 ID 以及用户的 Home 目录；

5）ClearKeepCaps，清除所有保留的 Linux Capability；

6）DropCapabilities，禁用其他所有的 Linux Capability；

7）syscall.Chdir(container.WorkingDir)，为容器进程切换至工作目录 workdir，即在用户态 docker run 命令指定的参数 workdir。

通过以上 7 个步骤，dockerinit 的 namespace 初始化工作圆满完成，一个备受瞩目的容器才真正诞生，因为容器的隔离、资源控制、权限管理等在这一刻才全部完成。dockerinit 在容器的诞生阶段扮演一个先行官的角色，然而，对于最终的用户而言，dockerinit 显然不是主角，用户希望自己指定的应用程序最终能在容器内按预期正常运行。

13.5.6　dockerinit 执行用户命令 Entrypoint

回到 namespaces 包的 Init 函数，在 FinalizeNamespace 之后，dockerinit 立即将容器主进程的执行权交给其他程序。而这些程序的启动命令是用户在 docker run 命令或者 Dockerfile 指定的 Entrypoint 命令与 Cmd 命令。

这部分代码位于 Init 函数的最后部分，如下：

```
return system.Execv(args[0], args[0:], os.Environ())
```

代码言简意赅，通过 Linux 进程的角度来分析再适合不过。首先，从 args 参数中提取出第一个参数作为执行命令，第二个开始所有的参数作为第一个执行命令的运行参数，而 os.Environ() 代表整个进程运行时的环境变量。同时，最为重要的一点是，通过系统调用执行 exec 操作，只是在原有进程的基础上重新执行一段程序，而并不会改变原有进程的 PID，也不会创建一个新进程。system.Execv 函数的定义位于 ./docker/libcontainer/system/linux.go#L11-L18，如下：

```
func Execv(cmd string, args []string, env []string) error {
    name, err := exec.LookPath(cmd)
    if err != nil {
        return err
    }

    return syscall.Exec(name, args, env)
}
```

在 Init 函数中，args 参数也是在 execdriver 在传递 createCommand 函数时被同时传递至 libcontainer 准备。Docker 体系中与 args 参数相关程度最高的当属 Entrypoint 与 Cmd，一旦用户指定了 Entrypoint 或者 Dockerfile 中指定了 Entrypoint，Docker Daemon 在处理用户命令参数时，就会将 Entrypoint 的内容放在 Cmd 前面，一起作为最终的 args；若均不存在 Entrypoint，那么 Docker Daemon 只将 Cmd 作为最终的 args。因此，对于一个 Docker 容器的配置而言，允许不存在 Entrypoint，但是必须要有 Cmd。在两者同时存在的情况下，启动容器时，Entrypoint 更偏向于容器应用层的初始化工作，在 Entrypoint 中的最后一个步骤，同样会使用 exec 系统调用，将主进程的执行权交给 Cmd，毋庸置疑，Cmd 属于用户指定运行的应用程序。

虽然容器的启动经历了 dockerinit、Entrypoint 以及 Cmd，但是这三者在运行时都基于同一个 PID，而整个容器的状态全部依赖于这个 PID 进程的存活。另外，当 Cmd 在运行过程

中创建其他子进程时，所有子进程都会受到相同 namespace 的作用，同时也会受到控制组等其他内核特性的作用。Cmd 进程及其所有子进程共同构成一个用户眼中的“容器”。

总之，Docker Daemon、dockerinit、Entrypoint 以及 Cmd 四者之间的关系如图 13-2 所示。

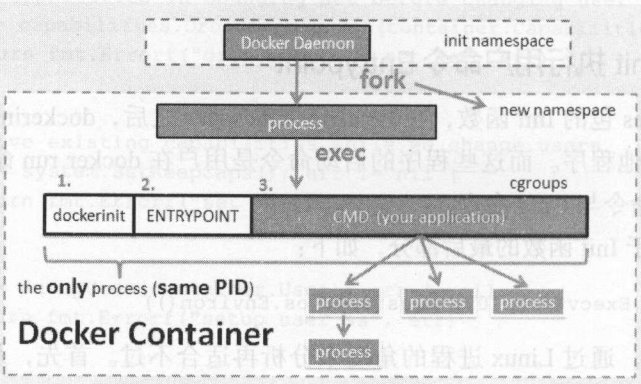
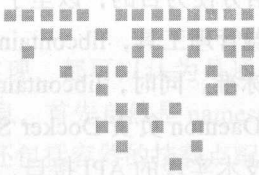


图 13-2 Docker Daemon、dockerinit、Entrypoint、Cmd 之间的关系

13.6 总结

深入学习 dockerinit 的原理，对于理解 Docker 容器的实现会有很大帮助。若对 dockerinit 思考得更多，则可以发现 dockerinit 的启动，即 Docker 容器的启动，完全是借助 Linux 内核的众多特性在做工作。从 Linux 内核的角度来看待 Docker 容器，容器技术的实现手段并非深不可测，容器毕竟和宿主机共享同一个 Linux 内核，所有工作都通过内核来完成。

Docker 容器的成功创建，可以说是 Docker Daemon 与 dockerinit 协同合作的结果。Docker Daemon 创建一个初步的容器环境，并将容器环境的初始化工作交给 dockerinit 来完成，同时两者之间完成起码的通信。dockerinit 的运行起到一个承前启后的作用，它不仅接受 Docker Daemon 的使命，完成容器环境的初始化，同时还负责将容器的运行转变为用户应用程序 Cmd 的运行，并将容器交付给 Docker 用户。



Chapter 14

第 14 章 libcontainer 介绍

14.1 引言

在 Docker 圈，大家也许听说过这样类似于这样的话“ Docker 的本质并非是新颖的技术，容器技术也早已诞生多年”。可以说，此言没有半点夸大。在 Docker 之前，Linux 平台上以 LXC 为代表的一系列容器技术早已在历史舞台上活跃多年。对于包括 Docker 在内的多种容器技术，最终的容器都离不开 Linux 内核的很多高级特性，如：namespace、cgroup 等。

Docker 架构中，Docker Daemon 作为一个常驻进程，管理 Docker Client 请求的同时，还管理所有的 Docker 容器。而 Linux 操作系统内核态对容器的管理，自然需要为用户态的 Docker Daemon 服务，因此如果在 Docker Daemon 与 Linux 内核之间有一套完善的 API 接口，那么两者的衔接将变得尤为顺畅。Docker 的发展历程中，0.9.0 版本即引入了 libcontainer 模块，这意味着内核 API 接口的横空出世。

简单而言，libcontainer 是一套实现容器管理的 Go 语言解决方案。这套解决方案实现过程中使用了 Linux 内核特性 namespace 与 cgroup，同时还采用了 capability 与文件权限控制等其他一系列技术。基于这些特性，除了创建容器之外，libcontainer 还可以完成管理容器生命周期的任务。

Docker 容器提供一个隔离、独立的环境，毫不过分地说，这样的效果完全离不开 libcontainer 的功劳。隔离的效果自然是容器内部与容器外部的隔离，然而，对于实现此功能的 libcontainer 而言，需要完成的工作正像容器内外一样，既需要在容器外做容器的配置与管理，同时还需要在容器内部完成相应的初始化工作。

另外，需要提及的是，libcontainer 作为一个提供内核 API 接口的容器技术解决方案，设

计之初就以简洁方便为目的，以至于 libcontainer 的生成与运行没有任何依赖。作为纯粹并且接口完备的容器管理工具，libcontainer 似乎有反向定义 Linux 容器技术的含义，尝试成为容器技术的新生标准。同时，libcontainer 的设计似乎也让开发者看到了 Docker 跨平台的潜在能力。Docker Daemon 负责 Docker Server 的一系列 API 接口，而 libcontainer 接管 Linux 平台内核态容器技术实现的 API 接口，两者通过 execdriver 的形式协调工作。如此一来，在原有 Docker Server 的 API 接口下，似乎替换其他平台的底层容器实现技术的接口即可，如在 Solaris 操作系统下，开发一个类似于 libcontainer 的库，兼容 Docker Daemon 的 API 接口，而实现 Solaris 的底层容器技术。因此，Docker 今后在跨平台方面的能力绝不可低估。

14.2 Docker、libcontainer 以及 LXC 的关系

Docker 的热潮席卷全球，没有适应于如此节奏的技术爱好者，或者曾在类似于 LXC 技术方面有所研究的开发者，肯定会如此发问：Docker 与 LXC 的区别到底在哪里？

从底层技术栈而言，似乎区别不大，Linux 的内核特性不仅 LXC 可以使用，Docker 可以使用，而且任何第三方的容器技术提供商均可以基于此进行开发。

从软件生命周期来讲，两者则存在很大的差异。Docker 可以这样认为：一个 Docker Daemon 管理众多的容器，Docker Daemon 为常驻的后台进程。而 LXC 则是一个工具，装有 LXC 的宿主机上并没有一个与 LXC 相关的常驻进程在后台运行。当用户发起与容器相关的创建与管理命令后，LXC 以进程的形式来处理命令，命令完成后进程立即退出。

既然 Docker 最终也需要创建与管理容器，那么是否也需要类似于 LXC 的工具呢？答案是肯定的。事实上，Docker 0.9.0 以前 Docker 正是通过 LXC 这套工具来完成底层容器的创建与管理；而从 Docker 0.9.0 开始 Docker 优先选择 libcontainer 作为创建以及管理容器的工具。Docker、libcontainer 以及 LXC 在 Docker 架构中的关系如图 14-1 所示。

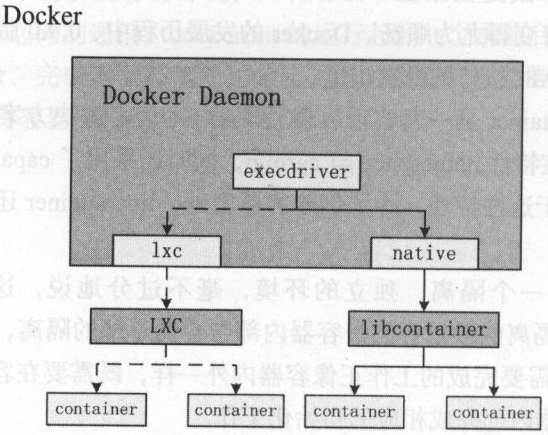


图 14-1 Docker、LXC 与 libcontainer 的关系

14.3　libcontainer 模块分析

Docker 容器范畴内一切与 Linux 内核相关的技术实现，都可以认为是通过 libcontainer 来完成的。libcontainer 指定了所有与容器相关的配置信息，首先自然是 namespace 与 cgroup 信息，除此之外，还有容器的网络栈配置信息，当然，还包括容器的挂载点配置、设备信息等。为实现与 Linux 内核的通信，libcontainer 还包含模块 netlink，完成内核态进程与用户态进程的 IPC（进程间通信）等。

本节主要介绍 libcontainer 的模块组成与各模块功能，对 libcontainer 的分析均基于 libcontainer 1.2.0 版本。

14.3.1　namespace

namespace 是 libcontainer 的左膀右臂，可以说没有隔离即不存在容器。对于一个容器而言，从它诞生的那一刻起，namespace 就已经开始为容器发挥作用了。Linux 操作系统中，创建进程一般会使用 fork 等系统调用。创建进程时，fork 等系统调用完成子进程对父进程 task_struct 的复制，且 fork 系统调用时可以传入众多与 namespace 相关的 flag 参数。而 Linux 操作系统通过 exec 来启动子进程的运行。回到容器的创建，由于容器的本质也是进程，故在派生进程时，Docker Daemon 传入了与 namespace 相关的 flag 参数，实现 namespace 的创建，确保容器（进程）被执行之后，也就是在进程在运行时达到 namespace 隔离的效果。

目前在 namespace 方面，libcontainer 有以下支持，源代码位于 ./docker/libcontainer/namespaces/types_linux.go#L7-L16，如下：

```
func init() {
    namespaceList = Namespaces{
        {Key: "NEWNS", Value: syscall.CLONE_NEWNS, File: "mnt"},
        {Key: "NEWUTS", Value: syscall.CLONE_NEWUTS, File: "uts"},
        {Key: "NEWIPC", Value: syscall.CLONE_NEWIPC, File: "ipc"},
        {Key: "NEWUSER", Value: syscall.CLONE_NEWUSER, File: "user"},
        {Key: "NEWPID", Value: syscall.CLONE_NEWPID, File: "pid"},
        {Key: "NEWNET", Value: syscall.CLONE_NEWNET, File: "net"},
    }
}
```

值得一提的是，虽然 namespace 中加入了用户命名空间，但是 libcontainer 由于一些特殊原因却并未将其完全实现。另外，对于网络命名空间的支持，也要视用户对容器的需求而定，若用户指定容器的网络模式为 host 模式，则 libcontainer 不为容器创建新的网络命名空间。

namespace 的完全生效，并不仅仅依靠 namespace 的创建，同时还需要对每一个 namespace 进行初始化。如 mount 命名空间、网络命名空间、uts 命名空间等命名空间需要 Docker Daemon 提供初始化信息，而其他命令空间则采用默认值。

1. namespace 的创建

为容器创建 namespace 完全是 Docker Daemon 的职责范畴，Docker Daemon 一方面从自身所在 namespace 创建新的 namespace 服务于容器，另一方面在容器 namespace 之外，为容器配置 namespace 之内所需的命名空间资源。libcontainer 的源码中，内部定义的 exec 函数实现了这部分内容，位于 ./docker/libcontainer/namespaces/exec.go#L24:

```
func Exec(container *libcontainer.Config, stdin io.Reader, stdout, stderr
io.Writer, console string, rootfs, dataPath string, args []string, createCommand
CreateCommand, startCallback func()) (int, error) {
    ...
    syncPipe, err := syncpipe.NewSyncPipe()
    ...
    command := createCommand(container, console, rootfs, dataPath, os.Args[0],
          syncPipe.Child(), args)
    ...

    if err := command.Start(); err != nil {
        return -1, err
    }
    ...
    started, err := system.GetProcessStartTime(command.Process.Pid)
    if err != nil {
        return -1, err
    }
    ...
    cgroupRef, err := SetupCgroups(container, command.Process.Pid)
    ...
    var networkState network.NetworkState
    if err := InitializeNetworking(container, command.Process.Pid, syncPipe,
        &networkState); err != nil {
        command.Process.Kill()
        command.Wait()
        return -1, err
    }

    state := &libcontainer.State{
        InitPid:       command.Process.Pid,
        InitStartTime: started,
        NetworkState:  networkState,
        CgroupPaths:   cgroupPaths,
    }

    if err := libcontainer.SaveState(dataPath, state); err != nil {
        command.Process.Kill()
        command.Wait()
        return -1, err
    }
    defer libcontainer.DeleteState(dataPath)
```

```
    // Sync with child
    if err := syncPipe.ReadFromChild(); err != nil {
        command.Process.Kill()
        command.Wait()
        return -1, err
    }

    ...

    return command.ProcessState.Sys().(syscall.WaitStatus).ExitStatus(), nil
}
```

Libcontainer 中 namespace 的 exec 实现，具体完成的主要工作有：

1）创建 syncpipe，以便后续 Docker Daemon 与容器进程跨 namespace 进行信息传递；

2）创建容器内部第一个进程的可执行命令；

3）启动该命令实现 namespace 的创建；

4）为容器的第一个进程进行 cgroup 的限制；

5）在 Docker Daemon 所在 namespace 中初始化容器内部所需的网络资源，以便后续通过管道的形式将资源传递至容器内部；

6）通过管道跨 namespace 将网络资源传递至容器进程。

2. namespace 初始化

Docker Daemon 将容器所需要的资源配置信息通过两种途径传递至容器，分别为：携带网络资源信息的 syncpipe 以及 Docker Daemon 持久化到宿主机的 container.json 文件。第 13 章分析过容器内第一个进程为 dockerinit，正是 dockerinit 完成了容器自身 namespace 内部挂载资源、用户资源、网络资源等的初始化工作。该部分内容实现位于 ./docker/libcontainer/namespaces/init.go#32-L122:

```
func Init(container *libcontainer.Config, uncleanRootfs, consolePath string,
syncPipe *syncpipe.SyncPipe, args []string) (err error) {
    ...
    if err := LoadContainerEnvironment(container); err != nil {
        return err
    }

    // We always read this as it is a way to sync with the parent as well
    var networkState *network.NetworkState
    if err := syncPipe.ReadFromParent(&networkState); err != nil {
        return err
    }
    ...
    if _, err := syscall.Setsid(); err != nil {
        return fmt.Errorf("setsid %s", err)
    }
```

```
    ...
    if err := setupNetwork(container, networkState); err != nil {
        return fmt.Errorf("setup networking %s", err)
    }
    ...
    if err := mount.InitializeMountNamespace(rootfs,
        consolePath,
        container.RestrictSys,
        (*mount.MountConfig)(container.MountConfig)); err != nil {
        return fmt.Errorf("setup mount namespace %s", err)
    }

    ...

    if err := label.SetProcessLabel(container.ProcessLabel); err != nil {
        return fmt.Errorf("set process label %s", err)
    }

    ...
    if err := FinalizeNamespace(container); err != nil {
        return fmt.Errorf("finalize namespace %s", err)
    }

    ...
    return system.Execv(args[0], args[0:], os.Environ())
}
```

容器的整个生命周期中，namespace 是最为基础的一块。Docker 容器的很多功能均在 namespace 的基础上完成。简单的理解是：隔离的环境创建完毕之后，内部环境的初始化才能登场。

14.3.2 cgroup

cgroup 是 Linux 内核的一个特性，此特性可以帮助用户对一组进程进行资源使用的限制、统计以及隔离。Docker 领域也不例外，Docker 利用对 cgroup 的支持，完成对 Docker 容器进程组的资源限制、统计以及隔离。

与 cgroup 内容相关的 Cgroup 定义，位于 ./docker/libcontainer/cgroups/cgroups.go#L40-L55，如下：

```
type Cgroup struct {
    Name   string
    Parent string                    // name of parent cgroup or slice

    AllowAllDevices   bool           // If this is true allow access to any
                                     kind of device within the container. If false,
                                     allow access only to devices explicitly listed
                                     in the allowed_devices list.
    AllowedDevices    []*devices.Device
```

```
    Memory           int64          // Memory limit (in bytes)
    MemoryReservation int64         // Memory reservation or soft_limit (in bytes)
    MemorySwap       int64          // Total memory usage (memory + swap); set
                                    `-1' to disable swap
    CpuShares         int64         // CPU shares (relative weight vs. other
                                    containers)
    CpuQuota         int64          // CPU hardcap limit (in usecs). Allowed cpu
                                    time in a given period.
    CpuPeriod         int64         // CPU period to be used for hardcapping (in
                                    usecs). 0 to use system default.
    CpusetCpus        string        // CPU to use
    Freezer           FreezerState  // set the freeze value for the process
    Slice             string        // Parent slice to use for systemd
}
```

从结构体 Cgroup 的定义可见，Docker 对于 cgroup 的支持，主要有以下 5 方面：设备（device）、内存（Memory）、CPU、Freezer 以及 systemd。在容器设备方面，Docker 支持让用户选择容器可以使用的设备。在内存方面，支持为容器的运行设定用量限额。在 CPU 方面，支持容器进程之间拥有相对的运行时间片。Freezer 则可以使容器挂起，节省 CPU 资源，Docker 命令中 pause 与 unpause 命令即采用了 cgroup 的 Freezer 子系统。需要注意的是，容器进程组挂起，并不意味着进程已经终止，从 Linux 内核的角度来看，被挂起的进程拥有完整的 task_struct，占用相应的内存，但是 Freezer 子系统保证容器中的进程不会被 CPU 调度到。Slice 属性则属于 systemd 方面的配置。

14.3.3　网络

Docker 的网络一直是一个备受关注的技术点。目前，关于容器网络的管理，底层实现全部借助 libcontainer 的 network 包来完成。network 包定义了 Docker 容器的网络栈类型。Docker 容器的网络模式有：bridge 模式、host 模式、other container 模式以及 none 模式。这 4 种网络模式对网络环境的配置分别如下。

- bridge 模式：为容器配置 veth 网络接口对，并配置 loopback 接口；
- host 模式：不为容器创建网络命名空间；
- other container 模式：不为容器创建网络命名空间，让容器与其他容器共享网络命名空间；
- none 模式：为容器创建网络命名空间，配置 loopback 接口。

libcontainer 为了标识网络接口的数据接口，抽象出 network 结构体，位于 ./docker/libcontainer/network/types.go#L7-L30，定义如下：

```
type Network struct {
    // Type sets the networks type, commonly veth and loopback
    Type string `json:"type,omitempty"`

    // Path to network namespace
```

```
    NsPath string `json:"ns_path,omitempty"`

    // The bridge to use.
    Bridge string `json:"bridge,omitempty"`

    // Prefix for the veth interfaces.
    VethPrefix string `json:"veth_prefix,omitempty"`

    // Address contains the IP and mask to set on the network interface
    Address string `json:"address,omitempty"`

    // Gateway sets the gateway address that is used as the default for the interface
    Gateway string `json:"gateway,omitempty"`

    // Mtu sets the mtu value for the interface and will be mirrored on both the
    host and
    // container's interfaces if a pair is created, specifically in the case of
    type veth
    // Note: This does not apply to loopback interfaces.
    Mtu int `json:"mtu,omitempty"`
}
```

Network 结构体代表容器的网络接口。容器网络接口类型（Type）一般有两种：veth 和 loopback；NsPath 代表容器网络命名空间的所在路径；Bridge 代表容器网络使用的网桥设备名；VethPrefix 代表容器 veth 网络接口名的前缀；Address 和 Gateway 分别代表容器网络接口的 IP 地址以及网关地址；最后，Mtu 则代表容器网络接口设备的 MTU 值。

14.3.4 挂载

文件系统的使用，Docker 容器绝不仅限于镜像所提供的 rootfs。volume 的存在（包括 bind-mount 以及 data volume）使得容器的文件系统可以共享宿主机的资源。除了 volume 之外，容器不少有关于容器信息的配置文件，也是通过挂载（mount）的形式使得容器可以使用 mount 命名空间之外的资源，这样的配置文件包括 hostname、hosts 以及 resolv.conf。

libcontainer 对于 Mount 对象的定义位于 ./docker/libcontainer/mounts/types.go#L28-L35，如下：

```
type Mount struct {
    Type        string `json:"type,omitempty"`
    Source      string `json:"source,omitempty"`
    // Source path, in the host namespace
    Destination string `json:"destination,omitempty"`
    // Destination path, in the container
    Writable    bool   `json:"writable,omitempty"`
    Relabel     string `json:"relabel,omitempty"`
    // Relabel source if set, "z" indicates shared, "Z" indicates unshared
    Private     bool   `json:"private,omitempty"`
}
```

对于 Docker Daemon 而言，记录 Mount 的源地址以及目标地址无疑是最重要的。在此基础上，再对 Mount 对象进行一些属性配置，如 Mount 对象是否开启容器的写权限等。

14.3.5　设备

对于容器而言，使用 Linux 内核管辖范围内的设备，是一个非常基本的需求。libcontainer 则使用 devices 包来为 Docker 容器提供相应的设备。

libcontainer 对于设备的定义位于 ./docker/libcontainer/devices/devices.go#L20-L27，如下：

```
type Device struct {
    Type              rune        `json:"type,omitempty"`
    Path              string      `json:"path,omitempty"`
    // It is fine if this is an empty string in the case that you are using Wildcards
    MajorNumber       int64       `json:"major_number,omitempty"`
    // Use the wildcard constant for wildcards.
    MinorNumber       int64       `json:"minor_number,omitempty"`
    // Use the wildcard constant for wildcards.
    CgroupPermissions string      `json:"cgroup_permissions,omitempty"`
    // Typically just "rwm"
    FileMode          os.FileMode `json:"file_mode,omitempty"`
    // The permission bits of the file's mode
}
```

默认情况下，libcontainer 会为容器创建某些必备的设备，如 /dev/null、/dev/zero、/dev/full、/dev/tty、/dev/urandom、/dev/console 等。

14.3.6　nsinit

libcontainer 的存在，使得 Docker 操纵 Linux 内核特性从而管理容器成为可能。既然如此，在逆向思维下，不少人肯定会有疑问：如果仅仅只有 libcontainer，而没有更多诸如 Docker Daemon 之类的软件，是否依然可以创建容器？

答案的是肯定的，nsinit 就是最好的例子。nsinit 是一个功能强大的应用程序，该应用程序巧妙利用了 libcontainer，使得容器的创建变得异常简单。若需要创建一个容器，则按照 libcontainer 的 API 接口，必须要向其提供容器的 rootfs 以及容器的参数配置。因此，nsinit 的运行需要提供一个 rootfs 以及一个容器的配置文件 container.json。JSON 文件 container.json 需要处于 rootfs 的根目录下，并且文件中含有的配置信息需要包括：容器的环境变量、网络和容器 Capability 等内容。

14.3.7　其他模块

libcontainer 的组成较为丰富，除了以上这些较为重要的组成部分之外，其他模块的存在也极大地完善着 libcontainer 的功能。

libcontainer 完成很多容器内部的配置工作，如何真正与 Linux 内核打交道，肯定是

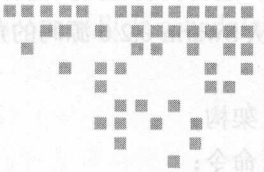
libcontainer 必须要完成的工作。Netlink 作为 Linux 内核的一套接口，提供进程间用户态与内核态之间的通信方式。

在容器的安全方面，libcontainer 由 security 模块负责。Linux 的 Capability 机制作为一项非常重要的安全指标，Docker 可以由用户来自定义配置容器的 Capability，具体实现由 security 包完成。

Capability 的存在极大地限制了用户进程的权限，然而，安全的范畴太广，不可能全部倚仗 Capability。由于 Docker 的 namespace 没有完全实现用户命名空间，因此 Docker 容器中的超级管理员用户 root 与宿主机上的 root 用户是同一个用户。没有隔离用户，的确是一个很大的安全隐患，然而，Docker 通过 Capability 特性尽量降低这方面的影响。可以说，使用了 Capability 特性之后，在权限方面，Docker 容器内的 root 用户和宿主机上的 root 用户有很大的差别，容器内 root 用户的权限自然会小很多。与此同时，Docker 还支持 SELinux 以及 Apparmor，为容器的安全保驾护航。

14.4 总结

开发者普遍了解甚至掌握 Docker，却往往会忽视 libcontainer 的重要性。libcontainer 作为容器技术的一种解决方案，很好地衔接了 Docker Daemon 这个管理引擎与 Linux 的内核特性。更为直接的评价是：Docker 的运行不能没有 libcontainer，而在 libcontainer 的基础上，任何开发者或团队都可以开发或者再造类似 Docker 的容器管理引擎。相信 Docker 官方也希望更多开发者参与 libcontainer 的维护工作，并努力推动其发展，使 libcontainer 成为容器技术的标准，从而在容器市场上占据更大的份额。



第 15 章 Chapter 15

Swarm 架构设计与实现

15.1 引言

Docker 自诞生以来，其容器特性以及镜像特性给 DevOps 爱好者带来了诸多方便。然而，在很长的一段时间内，Docker 只能在单宿主机上运行，其跨宿主机的部署、运行与管理能力颇受外界诟病。跨宿主机能力的薄弱，直接导致 Docker 容器与宿主机的紧耦合，这种情况下，Docker 容器的灵活性很难令人满意，容器的迁移、分组等都成为很难实现的功能点。

Swarm 是 Docker 公司在 2014 年 12 月初新发布的容器管理工具。和 Swarm 一起发布的 Docker 管理工具还有 Machine 以及 Compose。

Swarm 是一套较为简单的工具，用于管理 Docker 集群，使得 Docker 集群暴露给用户时相当于一个虚拟的整体。Swarm 使用标准的 Docker API 接口作为其前端访问入口，换言之，各种形式的 Docker Client（Go 语言的客户端、docker_py、docker 等）均可以直接与 Swarm 通信。Swarm 几乎全部用 Go 语言来完成开发，并且还处于一个 Alpha 版本。然而，Swarm 的发展十分快速，功能和特性的变更迭代还非常频繁。因此，可以说 Swarm 还不推荐用于生产环境中，但可以肯定的是 Swarm 是一项很有前途的技术。

Swarm 的设计初衷和其他 Docker 项目一样，遵循“batteries included but removable”原则。笔者对该原则的理解是：batteries included 代表设计 Swarm 时，为了完全体现分布式容器集群部署、运行与管理功能的完整性，Swarm 和 Docker 协同工作，Swarm 内部包含了一个较为简易的调度模块，以达到对 Docker 集群调度管理的效果；“but removable”意味着 Swarm 与 Docker 并非紧耦合，同时 Swarm 中的调度模块同样可以定制化，用户可以按照自己的需求，将其替换为更为强大的调度模块，如 Mesos 等。另外，这套管理引擎并未侵入

Docker 的使用，这套机制也为其他容器技术的集群部署、运行与管理方式提供了思路。

本章主要从 Swarm 0.2.0 源码的角度分析 Swarm 的架构设计与实现。分析内容包含以下三个步骤：

1）Swarm 架构；

2）Swarm 命令；

3）Swarm 运行。

15.2 Swarm 架构

Swarm 作为一个管理 Docker 集群的工具，我们必须首先将其部署于某一节点。Swarm 的部署形式比较丰富，我们可以将其部署于物理服务器上，也可以将其部署于虚拟机中，还可以将其部署在 Docker 容器中。

由于 Swarm 用于管理 Docker 集群，因此我们自然需要一个或者多个 Docker 集群，集群中每一个节点均安装并运行着 Docker。具体的 Swarm 架构如图 15-1 所示。

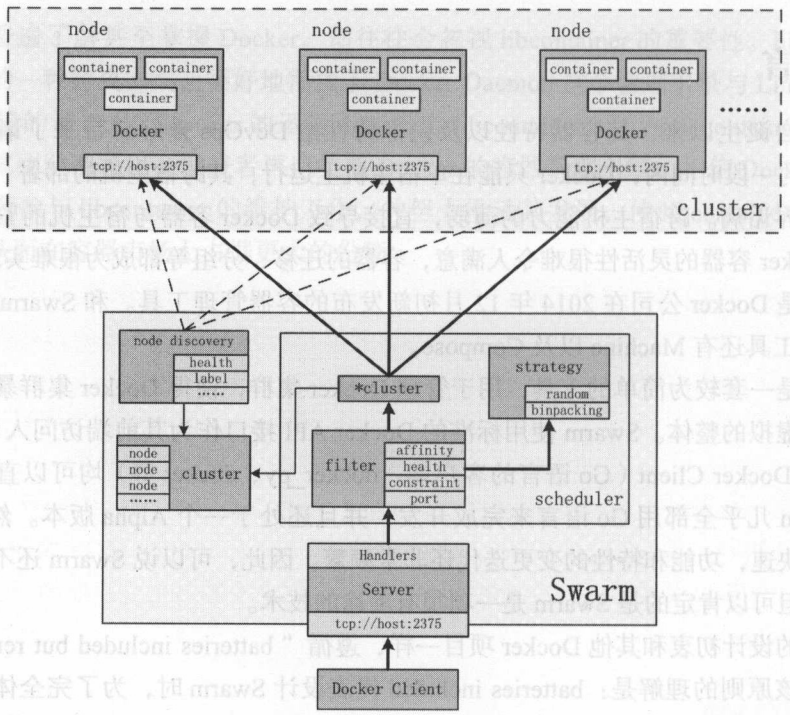


图 15-1 Swarm 架构

Swarm 架构中最主要的计算部分是 Swarm 节点，Swarm 管理的对象自然是 Docker Cluster，Docker Cluster 由多个 Docker Node 组成，而负责给 Swarm 发送请求的是 Docker Client。简单而

言，Swarm 与 Docker 一样，也是 C/S 架构，Client 为 Docker Client，Server 是 Swarm 进程。以下主要介绍 Swarm Node、Docker Node、node discovery 以及 scheduler 这 4 个重要的概念。

15.2.1　Swarm Node

Swarm Node 可以认为是 Swarm 的主控节点，角色由运行的 Swarm 程序来充当。Swarm Node 正常运行之后，该节点可以按需完成容器调度的任务，最终达到管理 Docker Node 集群的效果。

一般情况下，用户会通过添加标签（label）的形式，为 Docker Node 集群中的每一个节点设置角色。当需要调度 Docker 容器时，Swarm Node 根据用户需求以及所有 Docker Node 的角色，决策出最佳的 Docker Node 方案，并将容器调度至该 Docker Node。

为完成以上使命，Swarm 的设计者将 Swarm Node 设计成多个模块的形式。模块各司其职，并有机结合。Swarm 接收用户请求的模块暂且可以称为 Server 模块；用于发现集群中 Docker Node 的模块，我们可以将其抽象为 node discovery；收集 Docker Node 角色，处理请求，调度容器的任务则最终落在 scheduler 身上。对于 Swarm 而言，scheduler 的重要性不言而喻。更为细化地分析 scheduler，可以发现 scheduler 内部还有负责筛选 Docker Node 的 filter 模块，以及在筛选后的集合中如何做决策的 strategy 模块。

15.2.2　Docker Node

Swarm 管理 Docker 集群，Docker 集群则由一个或多个 Docker Node 组成。Docker Node 指的是运行 Docker Daemon 的计算节点。对于 Docker Node 而言，只要有足够的权限操纵 Docker Daemon，整个 Docker Node 的容器管理能力就被掌控。Swarm Node 就利用了这一点。Swarm 通过 tcp 的方式，访问 Docker Node 监听的 tcp 端口，从而控制 Docker Node。原则上，无论何时 Docker 管理员都应该尽量避免 Docker Daemon 监听 tcp 端口，以便带来安全隐患。特殊情况下，应该在 TLS 协议的保障下，开启 Docker Daemon 的 tcp 监听端口。

对 Docker Node 而言，与 Swarm Node 建立联系是其融入集群的第一步。然而，这还不够，还不能满足用户容器的按需调度。为实现有效合理的容器调度功能，Swarm 建议 Docker Node 使用标签的形式来标记自身的 Docker Daemon，使得 Swarm 获取 Docker Daemon 信息时记录 Docker Node 的角色。

15.2.3　node discovery

node discovery 属于 Swarm 架构中的服务发现机制。获知 Docker 集群中的 Docker Node 数量以及每个 Docker Node 具体的信息，关乎 Swarm 的管理效率以及集群的扩展能力。

服务发现的作用很明显，只有向 Swarm Node 注册过的 Docker Node 才会被 Swarm 划入 Docker 集群范畴并管理。Swarm 提供多种有效的服务注册方式，每当 Docker Node 注册完毕

之后，Swarm 有能力通过注册信息与 Docker Node 上的 Docker Daemon 进程建立通信，实现 Docker Node 的角色信息获取等。

15.2.4 scheduler

scheduler 属于 Swarm Node 的调度器。对于每一个容器的调度请求，scheduler 都有义务为其确定候选的 Docker Node，并对这些 Docker Node 按照用户指定的策略进行排序。目前，Swarm 0.2.0 中，scheduler 的策略有三种：spread、binpack 以及 random。其中，spread 和 binpack 策略会权衡每一个候选 Docker Node 的可用 CPU 资源、可用内存资源以及当前 Docker Node 上运行容器的数量。而 random 则简单得多，该策略不做任何计算，仅仅随机挑选一个候选 Docker Node。

15.3 Swarm 命令

通过 Swarm 架构图，大家可以对 Swarm 有一个初步的认识，比如 Swarm 的具体工作流程：Docker Client 发送请求给 Swarm；Swarm 处理请求并发送至相应的 Docker Node；Docker Node 执行相应的操作并返回响应。除此之外，Swarm 的工作原理依然还不够明了。

深入理解 Swarm 的工作原理，可以先从 Swarm 提供的命令入手。Swarm 命令主要有 4 个：swarm create、swarm manage、swarm join、swarm list。当然，还有一个 swarm help 命令，该命令有助于正确使用 swarm 命令，本章不再赘述。

15.3.1 swarm create

Swarm 中，swarm create 命令用于创建一个集群标志。当 Swarm 管理 Docker 集群时，Docker Node 通过这个全球唯一的集群标志实现节点注册功能，Swarm 通过该标志发现集群中的 Docker Node。

发起该命令之后，Swarm 会前往 Docker Hub 上内建的服务发现中获取一个全球唯一的 token，用于唯一地标识 Swarm 管理的 Docker 集群。

Swarm 的运行需要使用服务发现，目前该服务内建于 Docker Hub，该服务发现机制仍处于 alpha 版本，站点为：http://discovery-stage.hub/docker.com。

15.3.2 swarm manage

Swarm 中，swarm manage 是最为重要的管理命令。一旦 swarm manage 命令在 Swarm Node 上被触发，则说明用户开始使用 swarm 管理 Docker 集群。从运行流程的角度来讲，swarm 经历的阶段主要有两个：启动 swarm，接收并处理 Docker 集群管理请求。

Swarm 的启动过程包含三个步骤：

1）发现 Docker 集群中的各个 Docker Node，收集节点状态、角色信息，并监视节点状

态的变化；

2）初始化内部调度器模块；

3）创建并启动 API 监听服务模块；

第一步，Swarm 发现 Docker 集群中的节点。发现（discovery）是 Swarm 中用于维护 Docker 集群状态的机制。既然涉及发现，那么在这之前必须先有注册（register）。Swarm 中有专门负责发现的模块，而关于注册部分，不同的 discovery 模式下，注册也会有不同的形式。

目前，Swarm 中提供了 5 种不同的发现机制：Node Discovery、File Discovery、Consul Discovery、EtcD Discovery 和 Zookeeper Discovery。

第二步，Swarm 内部的调度器模块被初始化。当 swarm 通过发现机制发现所有注册的 Docker Node 并收集到所有 Docker Node 的状态以及具体信息后，一旦 Swarm 接收到具体的 Docker 管理请求，Swarm 就需要对请求进行处理，并通过所有 Docker Node 的状态以及具体信息，来筛选决策到底哪些 Docker Node 满足要求，并通过一定的策略将请求转发至具体的 Docker Node。

第三步，Swarm 创建并初始化 API 监听服务模块。从功能的角度来讲，可以将该模块抽象为 Swarm Server。需要说明的是：虽然 Swarm Server 完全兼容 Docker 的 API，但是有不少 Docker 命令目前是不支持的，毕竟管理 Docker 集群与管理单独的 Docker Daemon 会有一些区别。当 Swarm Server 初始化完毕并完成监听之后，用户即可以通过 Docker Client 向 Swarm 发送 Docker 集群的管理请求。

总之，Swarm 的 swarm manage 接收并处理 Docker 集群的管理请求，这也是 Swarm 内部多个模块协同合作的结果：请求入口为 Swarm Server，处理引擎为 scheduler，节点信息 Discovery。

15.3.3　swarm join

Swarm 的 swarm join 命令用于将 Docker Node 添加至 Swarm 管理的 Docker 集群中。从功能上出发，我们不难发现 swarm join 命令的执行位于 Docker Node，因此在 Docker Node 上运行该命令，首先必须安装 Swarm。由于 Docker Node 上的 Swarm 只可能执行 swarm join 命令，故我们可以将其看成是 Docker Node 上用于注册的 agent 模块。

就功能而言，可以认为 swarm join 完成 Docker Node 在 Swarm 节点处的注册（register）工作，以便 Swarm 在执行 swarm manage 时可以发现该 Docker Node。然而，15.3.2 节提及的 5 种 discovery 机制中，并非每种机制都支持 swarm join 命令。不支持的 discovery 机制有 Node Discovery 与 File Discovery。

Docker Node 上 swarm join 命令的执行，标志着 Docker Node 向 Swarm 注册。Swarm 通过注册信息，发现 Docker Node，并获取 Docker Node 的状态以及具体信息，以便处理 Docker 请求时作为调度依据。

15.3.4 swarm list

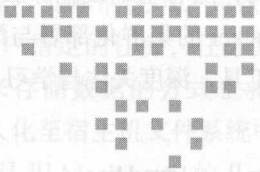
Swarm 中的 swarm list 命令用于列举 Docker 集群中的 Docker Node。

Docker Node 的信息均来源于 Swarm 节点上注册的 Docker Node。而一个 Docker Node 在 Swarm 节点上仅仅注册了 Docker Node 的 IP 地址以及 Docker Daemon 监听的端口号。使用 swarm list 命令时，需要指定 discovery 的类型，类型包括：token、etcd、file、zk 以及 <ip>。

15.4 总结

Swarm 的架构以及命令并不复杂，本章初步介绍了 Swarm、Swarm 架构的设计与实现以及 Swarm 命令，希望能为学习 Docker 集群化管理的 Docker 爱好者带来帮助。

俗话说得好，没有一种一劳永逸的工具，有效地管理 Docker 集群同样也是如此。脱离场景来谈论 Swarm 的价值，意义并不会很大。相反，探索和挖掘 Swarm 的特点与功能，并为 Docker 集群的管理提供一种可选的方案，则是 Docker 爱好者更应该参与的事。



第 16 章 Chapter 16

Machine 架构设计与实现

16.1 引言

2014 年 12 月，Docker 官方在荷兰举办的 DockerCon 大会上宣布推出 Docker Machine。此举预示着 Docker 从原先的单机部署迈向集群部署，Docker 生态圈的能力更上一层楼。

Docker 毕竟还是一个正在飞速成长的项目，尽管存在瑕疵，但是 Docker 在单机上的能力已经征服大量优秀的开发者。随着数十年来分布式系统的发展，分布式领域同样对 Docker 有着非常强烈的诉求。如何在服务器上自动化部署 Docker，如何跨宿主机部署，如何使 Docker 的部署适应不同的基础设施环境，都会是 Docker 在分布式生产环境中落地时必须考虑的因素。

正是在如此背景下，Machine 应运而生。Machine 使得 Docker 的部署异常简易，不论是用户的单个主机，还是用户的数据中心，以及可能是第三方云平台提供商提供的云主机。Machine 可以帮助用户在运行环境中创建虚拟机服务节点，在虚拟机中安装并配置 Docker，最终帮助用户配置 Docker Client，使得 Docker Client 有能力与虚拟机中的 Docker 建立通信。

一个大型的分布式环境中，Docker 集群的从无到有，Machine 甚至可以一键完成，这无疑将大大节省运维团队的人力、物力。同时，Machine 完全有能力适应多种不同的底层基础设施，如 OpenStack、VirtualBox、Amazon EC2、Azure、Rackspace、DigitalOcean、SoftLayer、Hyper-V 等一系列国际知名基础设施提供平台。另外，通过 Machine 创建的虚拟机，不仅已经完成 Docker 的安装与配置，同时还可以保证环境中所有 Docker 配置的一致性。只要是通过 Machine 创建并部署的 Docker 节点，Machine 就能对其进行远程管理或者执行 Docker 命令。

全球范围内，随着基础设施即服务（IaaS）越来越完善以及 Docker 的逐渐成熟，大型分布式环境中 Docker 的集群化部署与配置，是一个必须攻克的问题。Machine 作为 Docker 官方推荐的部署工具，深度关注与学习 Machine 将变得意义重大。

16.2 Machine 架构

Machine 可以帮助用户通过一条命令，从零开始，在极短的时间内，拥抱 Docker。为实现此目标，用户需要准备一些前提条件，如安装 Machine 软件（docker-machine）、提供第三方的虚拟机、提供软件或基础设施平台。

Machine 软件可以通过下载二进制文件获取，Docker 官方的下载地址为：http://docs.docker.com/machine/。官方提供的版本可以支持三种操作系统，即 Windows、OSX 以及 Linux。除此之外，用户可以通过编译 Machine 的源码生成 Machine 二进制文件，Machine 的源代码完全托管于 Github，地址为 https://github.com/docker/machine。目前，Machine 仍然处于 Beta 版本，功能以及特性都还在不断地变化中，因此 Docker 官方暂时还不建议将 Machine 运用于生产环境。Machine 的版本更新非常迅速，本章以 Machine v0.2.0 为例，分析 Machine 的架构以及实现。

Machine 的架构设计如图 16-1 所示。

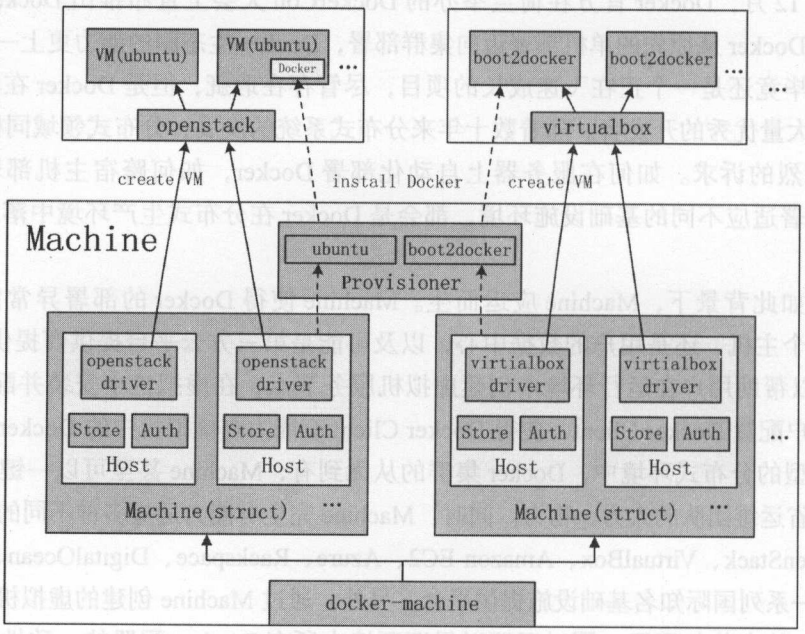


图 16-1 Machine 架构图

并非如 Docker 一般，Machine 不是一个 Client/Server 架构。用户通过 docker-machine 命

令使用 Machine 时，后台并非是一个常驻的服务进程。实际情况下，每次 docker-machine 命令被触发时，系统都会创建一个相应进程来完成用户指定的任务，任务完成后进程随即退出。由于 Machine 并没有后台进程，故通过内存来存储数据的方式也将行不通，此时 Machine 将所有创建的虚拟机信息以及 Docker 信息持久化至宿主机文件系统中。

为了更好地理解 Machine 的架构，首先我们来认识 Machine 中的几个重要的概念：Machine、Store、Host、Driver 以及 Provisioner。

16.2.1　Machine

此 Machine 并非代表 Machine 软件，也不指代 docker-machine 二进制文件，而是软件实现过程中抽象出的一个数据结构。

对于 docker-machine create 命令而言，一个 Machine 对象会创建，然后所有与该 Machine 相关的信息都存储到指定的路径中。

16.2.2　Store

对于每一个创建的 Machine 对象，相应的元数据都会被持久化到本地文件系统中，Store 则用于告知用户这些元数据的存储位置。Store 类型包含三个属性：root、caCertPath、privateKeyPath。属性 root 代表 Machine 对象信息存储的根目录；caCertPath 代表连接 Machine 所需证书的路径；而 privateKeyPath 则代表通过 ssh 连接虚拟机时私钥所在路径。

16.2.3　Host

Host 代表通过 Machine 在相应的基础设施上创建的虚拟机。每一个 Host 对象都包含 Machine 管理一台虚拟机所需要的所有信息。这些信息包括虚拟机名称（Name）、连接基础设施的 driver 名称（DriverName）、具体的 Driver 对象（Driver）、存储 Machine 实例所有信息的路径（StorePath）、包含 Host 具体信息的选项（HostOptions），另外还有与 Swarm 相关的信息以及认证内容。

值得一提的是 HostOptions，其携带的虚拟机信息都在整个 Machine 中扮演重要角色。首先，包括虚拟机的内存大小、磁盘大小。另外，HostOptions 的 EngineOptions 属性携带虚拟机中 Docker Daemon 的配置信息，包括：DNS 配置、镜像存储 driver、Daemon Label 等众多配置信息；SwarmOptions 属性代表是否为该虚拟机中创建一个 Swarm 节点；AuthOptions 则是认证信息，用于建立用户与 Docker Daemon 之间的通信。抽象 Machine 源码，归纳总结出的 Host 数据结构如图 16-2 所示。

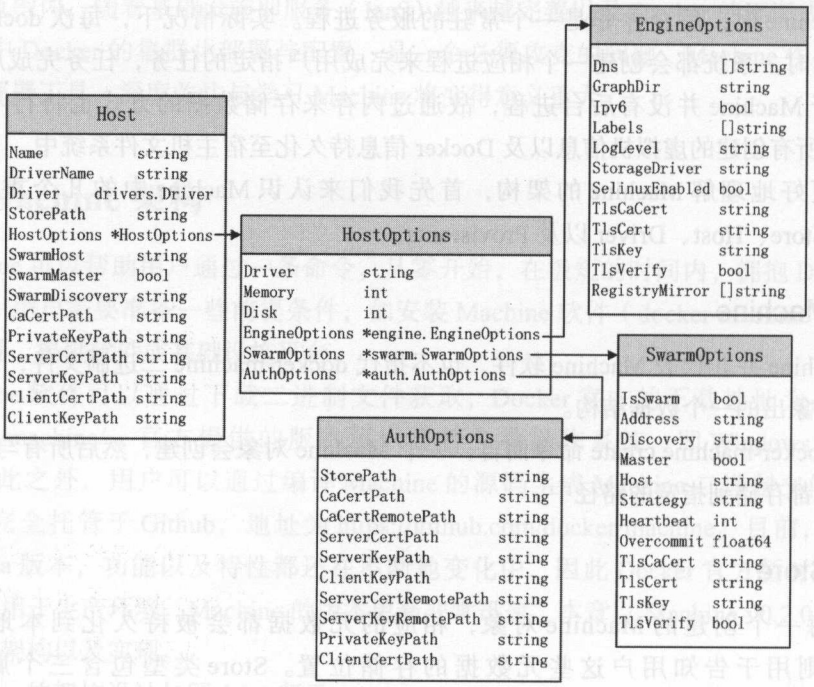


图 16-2 Host 数据结构

16.2.4 Driver

Driver 在 Machine 中起到承前启后的作用，承前的是：所有创建虚拟机的请求，最终都需要落实到指定的 Driver 层；启后的是：Driver 向具体的基础设施发起虚拟机创建请求，基础设施的响应也将首先返回至 Driver。

16.1 节提到，Machine 目前支持众多的基础设施供应商。如此一来，由于不同基础设施平台之间存在对外接口的差异性，Machine 很难做到提供一个大而全的 Driver，涵盖所有基础设施，因此 Machine 中 Driver 的种类会与基础设施一一对应。事实的确如此，对于不同的基础设施，Machine 使用不同的 Driver 来完成任务，openstack driver 则完全适配 OpenStack 平台的接口，完成 OpenStack 平台上虚拟机的创建；对于 VirtualBox 这种虚拟化技术，Machine 采用 virtualbox driver，进而使用 VBoxManage 管理工具完成 VirtualBox 下虚拟机的申请……

16.2.5 Provisioner

Driver 完成的工作是：在基础设施平台上从无到有创建虚拟机。然而，整个过程中并不涉及 Docker、Docker Daemon 的内容，换言之，创建虚拟机之后，虚拟机中并没有安装 Docker。Provisioner 对象则起到这最后一块拼图的作用。

Machine 的使命就是在裸环境中创建装有 Docker 的机器，并统一化管理。因此，通过 Driver 创建虚拟机之后，Provisioner 完成 Docker 安装与配置的任务。不同虚拟机有可能操作系统类型不同，而操作系统不同，Provisioner 采取的手段也不同。目前，Machine 仅仅支持两种操作系统发行版的 Docker 安装，分别为 Ubuntu 和 boot2docker。

一次完整的 Provision 流程，包含以下 5 个步骤：

1）为虚拟机设置主机名 hostname；

2）若虚拟机中没有安装 Docker，则为其安装 Docker；

3）配置 Docker Daemon 参数，使其启动后仅仅接受 TLS 连接；

4）复制证书至本地 Machine 实例的目录，并上传证书以及 TLS 认证信息至虚拟机；

5）若用户指定 Machine 与 Swarm 的结合，通过 Docker 启动 Swarm 容器，并对其进行配置。

16.2.6　Machine 运行流程

清楚 Machine 中重要的概念之后，我们一起进入 Machine 的运行流程一探究竟。回到图 16-1 的 Machine 架构图中，假设现在用户的需求是：在 OpenStack 平台上创建一台虚拟机，虚拟机的操作系统为 Ubuntu，并且虚拟机中必须安装有 Docker。我们以此为例，分析 Machine 的运行流程。

Machine 的运行流程主要围绕 Machine、Host、Provisioner 展开，如下：

1）用户通过 docker-machine 运行用户命令。该命令中必须要包含 openstack driver 的名称，以及为所创建虚拟机指定的名称，命令的示例如下：docker-machine create -d openstack dev ……

2）Machine 解析 Create 命令，配置 Store 参数，检测 driver 类型，创建 Machine 对象，并配置 hostOptions 参数；

3）Machine 通过虚拟机指定的名称，用户指定的 driver，以及虚拟机配置信息 hostOptions 创建 Host 对象；

4）Machine 将创建的虚拟机设置为 active 状态。

以上四个环节中，创建 Host 的环节涉及内容最多。Machine 创建虚拟机首先需要获知 driver 类型，即用户需要在哪种基础设施平台上创建。解析 -d 参数，仅仅获取 driver 类型 openstack 还远远不够，用户还需要指定与 openstack 平台相关的诸多信息，如代表 OpenStack 认证 URL 的 OS_AUTH_URL，OpenStack 的用户名 OS_USERNAME 和密码 OS_PASSWORD，OpenStack 平台中镜像的 ID“ openstack-image-id”等。由于前面假设通过 Machine 创建 Ubuntu 虚拟机，故指定 ID 所代表的镜像应该为 Ubuntu 操作系统。

OpenStack driver 解析出所有参数之后，通过 driver 内置的 openstack client 向基础设施平台发起虚拟创建请求。创建完毕之后，也是 Provisioner 登场之时。Provisioner 的第一个任务便是获悉虚拟机内部的操作系统发行版类型（Ubuntu 或者 boot2docker），实现方式为：通过

SSH 远程登录虚拟机，并查看 /etc/os-realease 的内容，以此判断操作系统发行版的类型。类型的确定，意味着 Provisioner 可以对虚拟机实行有针对性的 Docker 安装计划，具体可以参见前面 Provisioner 的执行流程。

完成虚拟机的创建，完成 Docker 的安装，原则上来讲，创建工作已经完成。然而，为了让用户方便地使用新创建的 Docker，Machine 可以通过命令 eval "$(docker-machine env dev)"，使得本地的 Docker Client 无需显式指定即可连接远程的 Docker Daemon。

16.3 Machine 与 Swarm 的结合

Machine 在 Docker 体系中的作用是：创建装有 Docker 的虚拟机。通过 Machine 强大的部署能力，我们并不能很好地调度和管理 Docker 集群。而 Swarm 的作用正是为 Docker 集群进行有效的管理与调度，若能将 Machine 与 Swarm 有效结合，必能为 Docker 集群的管理人员带来巨大的便利。

Machine 的设计理念则充分考虑了这一点。只要用户有需求，Machine 就完全有能力在虚拟机中安装 Docker 时，通过 Docker Daemon 启动一个 Swarm Master 节点，用来管理其余的 Docker Node。当然，Machine 也可以在安装 Docker 时，启动一个 Swarm Agent 容器，并将 Docker Daemon 作为一个 Docker Node 注册到指定的 Swarm Master 中，便于 Swarm Master 后续对于该 Docker Node 的调度。Swarm Master 并未直接运行在虚拟机中，而是运行在 Docker 容器中。Machine 与 Swarm 的关系如图 16-3 所示。

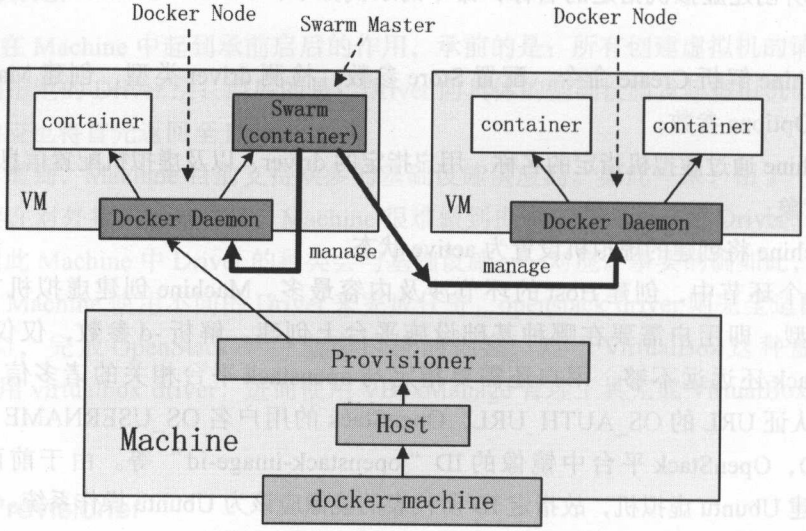


图 16-3 Machine 与 Swarm 的关系

关于如何在虚拟机上安装并配置 Docker，16.2 节已经讲述得较为清晰。事实上，Swarm

的引入完全是在此基础上完成的。由于 Swarm 的运行环境为 Docker 容器，故 Swarm 的启动离不开 Docker。实际运行过程也是如此，若用户指定安装 Swarm，则 Machine 首先通过 Provisioner 在虚拟机中安装 Docker，并在 Provisioner 安装与配置的最后一个步骤执行 configureSwarm，完成 Swarm 的启动。

若需要配置 Swarm，不论是 Swarm Master 还是 Swarm Agent，用户均需在执行 docker-machine create 命令时传入相应的参数。

首先来看 Swarm Master 的配置流程。Docker 安装完毕之后，Machine 中的 Porvisioner 模块根据用户的需求，需要在虚拟机上配置 Swarm Master，故 Provisioner 通过 SSH 连接至虚拟机中，并执行 docker pull swarm:latest 请求，用于下载 swarm 镜像。值得一提的是，swarm 镜像不仅可以用来启动 Swarm Master，同样可以用来启动 Swarm Agent。由于用户指定启动 Swarm Master，故 Provisioner 又执行 docker run swarm:latest manage……命令，直接启动 Swarm Master，并管理传入的 Docker Node。

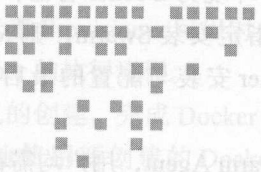
一般而言，一旦用户要求配置 Swarm，不论配置 Swarm Master 还是 Swarm Agent，虚拟机都会配置 Swarm Agent，并受到 Swarm Master 的管理。如此一来，会出现 Swarm Master 与 Swarm Agent 处于同一个 Docker Node 上，而这种情况其实并不矛盾，Swarm Master 也可以很好地管理自身所在的 Docker Daemon。对于 Swarm Agent 的配置，Provisioner 则执行 docker run swarm:latest join……命令，使得 Swarm Agent 注册到指定的 Swarm Master。

Swarm Master 与 Swarm Agent 的配置完毕，则意味着 Swarm 集群的成功创建。Machine 的存在很大程度上降低了用户与基础设施打交道的成本，从 0 到 1 快速创建 Swarm 集群。

16.4　总结

面对大型分布式环境或者尚未利用的基础设施，如何快速、有效、方便地使用 Docker，已经不再是一个脱离现实的问题。某种程度上讲，Machine 的诞生使得 Docker 技术的普及超越了开发者的单机模式，更是为中大型企业大规模运用 Docker 提供了可能性。

然而，对于不同的场景，Machine 的价值也不尽相同，快速创建 Docker 并分发虚拟机的模式，在大规模的开发环境下，能为开发者带来很大的价值。Machine 与 Swarm 甚至其他容器调度引擎的结合，更是为大规模集群的管理与调度提供了莫大的参考价值。另外，Machine 对多基础设施的支持，同样大大降低了云平台迁移的难度，为混合云的发展，提供强有力的事实依据。



Chapter 17

第 17 章 Compose 架构设计与实现

17.1 引言

众所周知，随着不断地发展与完善，Docker 的 API 接口变得越来越多。尤其在容器参数的配置方面，功能的完善势必造成参数列表的增长。若在 Docker 的范畴内管理容器，则唯一的途径是使用 Docker Client。而 Docker Client 最原生的使用方式是：利用 docker 二进制文件发送命令行命令。一些特殊的应用场景下，容器管理过程的配置项极为冗长，甚至很可能是多容器的环境。因此，通过命令行来完成容器管理显然不是长久之计。很长一段时间内，全球的 Docker 爱好者都在探索以及寻找方便容器部署的途径。

Docker 诞生于 2013 年 3 月，同年 12 月，基于 Docker 容器的部署工具 Fig 隆重登场。在 Docker 生态圈中，经过了两年多的洗礼，Fig 项目得到飞速发展的同时，背后的东家也发生了很大的变化。作为 Docker 界容器自动化部署工具的翘楚，Fig 原本是英国伦敦一家创业型公司的产品。随着产品的发展，Fig 的巨大潜力受到工业界的普遍认可，在不到一年的时间内就受到 Docker 公司的密切关注。很快就在 2014 年 7 月双方爆出新闻：Docker 收购 Fig。收购完成之后，Fig 改名为 Compose，命令改为 docker-compose。

17.2 Compose 介绍

探听 Fig 与 Compose 的前世今生之后，让我们回到 Compose 本身，尝试挖掘 Compose 如何在 Docker 乱世中崭露头角，尝试分析 Compose 的技术以及定位又是如何。

认识一样新事物，从新事物的作用入手，往往不会出太大差错。而 Compose 最大的作

用，就是帮助用户缓解甚至解决容器部署的复杂性。最原始的情况下，通过 Docker Client 发送容器管理请求，尤其是 docker run 命令，一旦参数数量骤增，通过命令行终端来配置容器较为耗时，同时容错性较差，且修复错误命令的时间成本很高。Compose 则将所有容器参数通过精简的配置文件来存储，用户最终通过简短有效的 docker-compose 命令管理该配置文件，完成 Docker 容器的部署。

编辑配置文件与编辑命令行命令的难易程度高下立判。同时配置文件数据的结构化程度越高，可读性也会越强。传统情况下，如 docker run 等命令的参数数量很多时，由于 flag 参数的书写格式各异，很容易造成用户费解的情况；而配置文件中一行内容就是一类具体的参数值，可读性大大增强。

在生产环境下，Docker Client 还有一方面经常被 Docker 爱好者所诟病，那就是难以进行多容器的管理，每次管理的容器对象最多只能是 1 个。容器虽然运行时相对非常独立，但是很多情况下，容器之间会存在逻辑关系，如容器 A 使用容器 B 的 data volume，如容器 C 需要对容器 D 执行 link 操作等。对于有逻辑关联的容器，如果能将其作为一个整体，被工具统一化管理，那将大大减少用户的人为参与，提高部署效率。

诱人的功能与软件的完美之间，往往不能画等号，Compose 同样如此。毕竟 Compose 的调用对象为 Docker，故 Docker 的发展将直接影响到 Compose 的未来。Docker 尚且还没有达到完美的地方，更遑论 Compose，因此 Docker 官方并不建议 Compose 爱好者在生产环境中使用该工具。除此之外，Compose 本身也存在一些缺陷，不熟悉其本质，自然也会深陷其中，难以脱身。

Compose 软件的开发绝大部分通过 Python 语言完成，而本章的分析均基于 Compose 1.2.0 版本。

17.3　Compose 架构

Docker 生态圈中，Compose 扮演的是部署工具的角色。用户使用 Compose 时，首先需要将部署意图通过配置文件的形式交给 Compose。这样的配置需求包括：容器的服务名、容器镜像的 build 路径、容器运行环境的配置等。以下是一个较为简单的 Compose 配置文件。此配置文件定义了两个服务，名称分别为 web 以及 db。服务 web 的镜像可以通过 docker build 来构建，Dockerfile 所处目录为该配置文件当前目录；服务 web 需要对 db 服务进行 link 操作；最终服务 web 将宿主机上的 8000 端口映射到容器内部的 8000 端口。服务 db 通过镜像 postgres 来创建。

```
web:
  build: .
  links:
   - db
  ports:
```

```
    - "8000:8000"
  db:
    image: postgres
```

配置文件在 Compose 体系中不可或缺。Fig 时代支持的配置文件名为 fig.yml 以及 fig.yaml；为了兼容遗留的 Fig 化配置，目前 Compose 支持的配置文件类型非常丰富，主要有以下 5 种：fig.yml、fig.yaml、docker-compose.yml、docker-compose.yaml 以及用户指定的配置文件路径。

配置文件的存在为 Compose 提供了容器服务的配置信息，在此基础上，Compose 通过不同的命令类型，将用户的 docker-composer 命令请求分发到不同的处理方法进行相应的处理。用户 docker-compose 的命令类型有很多，如命令请求 docker-compose up……的类型为 up 请求，Compose 将 up 请求分发至隶属于 up 的处理方法来处理；命令请求 docker-compose run……的类型为 run，Compose 将 run 请求分发至隶属于 run 的处理方法来处理。

对于不同的 docker-compose 请求，Compose 将调用不同的处理方法来处理。由于最终的处理必须落实到 Docker Daemon 对容器的部署与管理上，故 Compose 最终必须与 Docker Daemon 建立连接，并在该连接之上完成 Docker 的 API 请求。事实上，Compose 借助 docker-py 来完成此任务。docker-py 是一个使用 Python 开发并调用 Docker DaemonAPI 的 Docker Client 包。需要说明的是，毕竟 docker-py 作为 Docker 官方的一个 Python 软件包，和 Docker 并不隶属于同一个项目，因此 docker-py 在很多方面的发展均会滞后于 Docker，即理论上而言，docker-py 支持的 API 接口理论上会比 Docker Daemon 原生支持的 API 接口要少。因此，当使用 docker-py 作为 Docker Client 访问 Docker Daemon 时，确定 API 版本的支持是非常有必要的一步。另一方面，在 docker-py 支持的 Docker API 接口之中，Compose 也并没有对其进行百分之百的实现，而这主要受限于 Compose 自身的软件定位。

清楚 Compose 的配置文件、处理方法以及 docker-py 概念之后，再来分析 Compose 的架构，如图 17-1 所示。

在 Compose 架构中，我们可以发现三个新的部分，分别为：project、service 以及 container。这三个概念均为 Compose 抽象的数据类型，其中 project 会包含 service 以及 container。首先介绍这三者的意义。

project 代表用户需要完成的一个项目。何为项目？ Compose 的一个配置文件可以解析为一个项目，即 Compose 通过分析指定配置文件，得出配置文件所需完成的所有容器管理与部署操作。例如：用户在当前目录下执行 docker-compose up -d，配置文件为当前目录下的配置文件 docker-compose.yml，命令请求类型为 up，-d 为命令参数，对于配置文件中的内容，Compose 会将其解析为一个 project。一个 project 拥有特定的名称，并且包含多个或一个 service，同时还带有一个 Docker Client。

Service，代表配置文件中的每一项服务。何为服务？即以容器为粒度，用户需要 Compose 所完成的任务。再次以本节前面配置文件为例，配置文件中共包含了两个 service，

第一个名为 web，第二次名为 db。一个 service 包含的内容，无非是用户对服务的定义。定义一个服务，可以为服务容器指定镜像，设定构建的 Dockerfile，可以为其指定 link 的其他容器，还可以为其指定端口的映射等。

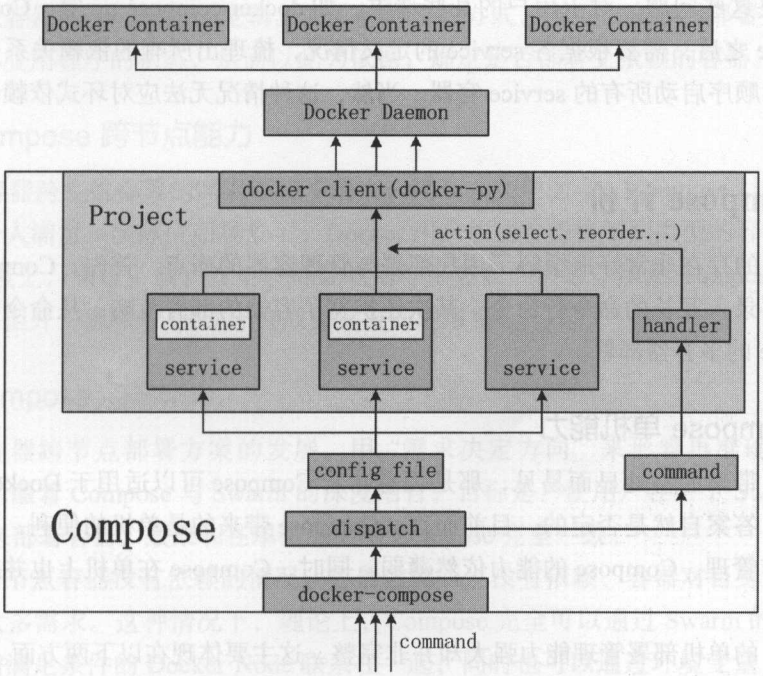


图 17-1　Compose 架构

从配置文件到 service，实现了用户语义到 Compose 语义的转换。虽然一个 service 尽可能详细地描述了一个容器的具体信息，但是 Compose 并非一定必须在 service 之上管理容器，如果用户使用 docker-compose pull db 命令，则仅仅完成 db 服务中指定镜像的下载。除此之外，Compose 的 service 还可以映射到多个容器，如果用户使用 docker-compose scale web=3 命令，则可以将 web 服务横向扩展到 3 个容器。

若用户对 service 的请求最终会落实到一个具体的容器上，则 Compose 会在 service 范畴内创建一个 container 对象，完成对具体容器的管理。container 对象初始化时即集成了 service 的 Docker Client，最终容器所有的操作，都由 container 对象通过调用 Docker Client 完成。

读到这里，大家可能会一个疑惑：Compose 如何决策所有 service 的执行顺序？如果 Compose 一味按照配置文件中的书写顺序来完成 service 的指定任务，显然会出现一些不可避免的问题。假设多个 service 所描述的容器之间存在依赖关系，一旦配置文件中的顺序与实际的正常启动顺序不一致，必将导致容器启动失败。若在配置文件中容器 A 的描述位于容器 B 之前，而容器 A 的启动又依赖于容器 B，此时若顺序执行 A、B，A 容器的启动必定将失败，而之后 B 容器可以正常启动。

一般而言，容器依赖关系会存在以下三种情况：存在 links 参数，容器的启动需要链接到另一个容器；存在 volumes_from 参数，容器的启动需要挂载另一个容器的 data volume；存在 net 参数，容器的启动过程中网络模式采用 other container 模式，使用另一个容器的网络栈。为了解决这些问题，对于用户的某些请求，如 docker-compose up 等，Compose 在解析出所有 service 之后，需要根据各 service 的定义情况，梳理出所有的依赖关系，并最终以一个没有冲突的顺序启动所有的 service 容器。当然，这种情况无法应对环式依赖。

17.4 Compose 评价

Compose 的存在非常好地缓解了用户部署与管理容器的痛点。首先，Compose 的存在使得用户无须再录入冗长的命令行命令，其次还扩宽了容器的部署范畴：从命令行的单容器部署到 Compose 的多容器部署。

17.4.1 Compose 单机能力

Compose 带来的益处显而易见，那是否意味着 Compose 可以适用于 Docker 容器部署的所有场景呢？答案自然是否定的。目前而言，Compose 带来的是单机的便利，面对跨宿主机的容器部署与管理，Compose 的能力依然薄弱。同时，Compose 在单机上也并非可以完成所有的工作。

Compose 的单机部署管理能力强大却并非完整。这主要体现在以下两方面。

首先在架构上，如前所述，Compose 的能力会受限于两个方面：自身的设计理念以及 docker-py 对 Docker API 支持的完整程度。既然 Compose 定位为 Docker 容器的部署工具，那么只要自足于这个理念，它就不可能支持所有的 Docker API 接口，否则从功能上将会变得类似于第二个 docker-py。然而，既然 Compose 中包装了 service 的概念，那么除了查看的 service 的状态之外，更为细致的情况下还应该能够查看 service 的资源使用情况。这个方面，Compose 目前还没有集成，毕竟 docker stats 命令在 Docker 1.5.0 版本时才支持，Compose 对这些 API 的支持显得有些滞后，docker-py 的使用也是造成这类问题的重要原因。

其次在使用上，Compose 的自动化部署并不意味着可以完全替代手动管理。还以容器依赖为例，虽然 Compose 目前支持判断容器间的依赖，并生成合理无冲突的执行顺序，但是这样的执行顺序，仅仅在时序上有区分，并未在容器应用的逻辑上区分。例如，容器 A 为一个 Web 应用，依赖于容器 B 这个数据库，配置文件 service A 的 links 参数指明了 service B，则 Compose 会在容器 B 启动完成之后再启动容器 A。这看似合理，实则不然。对于容器 A 而言，应用程序需要创建到数据库的连接，一旦失败，它可能直接退出。而数据库容器 B 的启动需要一个过程，只要 dockerinit 进程开始执行，Docker Daemon 均会认为该容器已经正常启动，故 Compose 通过 Docker Client（docker-py）也会认为容器 B 已经启动，从而立即启动容器 A。由于此时容器 B 中的数据库应用仍需要完成初始化，直至最终数据库服务引擎完全

启动之后，才能监听特定端口，接受外界应用的连接。B 容器需要一定时间来完成这个阶段，而 Docker Daemon 不会理会容器内部的应用逻辑，直接认为容器启动完毕，最终 Web 应用 A 无法连接 B 中的数据库服务，自然会异常退出。总的来说，Compose 还是立足于容器层的部署，并不涉及容器内部应用层的逻辑。而手动部署的方式，则可以在容器部署完毕时，人为测试容器内部应用程序的状态，从而以此为依据，确定是否部署受依赖的容器。

17.4.2　Compose 跨节点能力

Docker 容器跨主机部署的需求正逐步增大，稍令人失望的是，Compose 目前在这方面的功能依旧不令人满意。实际应用场景下，Docker 用户往往希望将不同类型的容器部署在不同的 Docker 节点上，满足负载、安全、资源利用等多方面的考虑。虽然 Compose 目前不具备这样的能力，但并不意味着 Docker 会放弃这方面的市场。

17.4.3　Compose 与 Swarm

Docker 容器跨节点部署方案的发展，用“需求决定方向”来形容再准确不过。目前，Docker 正在酝酿着 Compose 与 Swarm 的深度结合，目标是：使用户在一个 Swarm 集群上运行 Compose 来部署容器，效果和在单机上使用 Compose 完全一致。

先分析跨节点容器没有依赖的情况。容器之间一旦没有依赖，容器对自身所处的节点位置也就没有太多需求。这种情况下，理论上，Compose 完全可以通过 Swarm 的 label 环境变量，将容器与满足条件的 Docker Node 联系在一起；同时也可以通过环境变量 affinity，使几个容器部署在同一个 Docker Node 上或者避免在同一个 Docker Node 上。

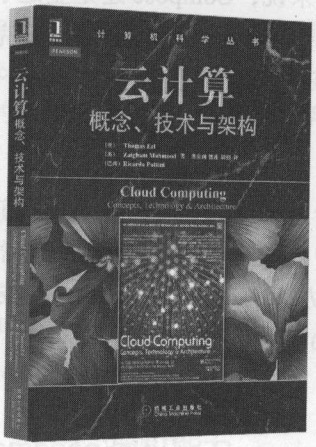
再研究跨节点容器存在依赖的情况。跨节点容器有依赖，第一个需要解决的问题是跨节点容器的通信能力。而在 Docker 的范畴内，如果不借助其他工具，跨节点容器的通信目前还没有很好的支持。因此，如 links、volumes_from、net:container 等容器依赖的情况，下一版本的 Compose 还会默认将相应的容器在同一个主机上部署运行。

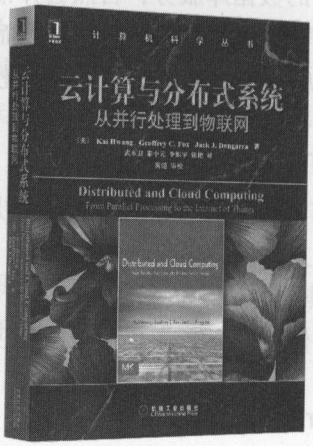
17.5　总结

本章从 Compose 的历史与定位入手，随后简单分析架构设计与内部实现，最后评价了 Compose 在容器部署方面的能力。

Compose 的诞生，并不是为了解决一切部署问题。如果信服这一点，那么研究 Compose 的应用场景就变得极为有价值。什么样的场景下，可以借助 Compose 发挥自动化部署的魅力，什么样的场景下，Compose 并不能满足需求，而需要用户自行开发工具进行部署与管理，才是在 Docker 容器部署与管理领域值得深究的话题。Docker 生态圈仍需要不断地发展，在容器缺乏部署工具方面，Compose 仍然会处于一枝独秀的地位。清楚 Compose 的原理，对于 Compose 的运用定会有很大的帮助。

推荐阅读





云计算：概念、技术与架构

作者：Thomas Erl 等 译者：龚奕利 等 ISBN：978-7-111-46134-0 定价：69.00元

“我读过Thomas Erl写的每一本书，云计算这本书是他的又一部杰作，再次证明了Thomas Erl选择最复杂的主题却以一种符合逻辑而且易懂的方式提供关键核心概念和技术信息的罕见能力。”

—— Melanie A. Allison，Integrated Consulting Services

在本书中，世界上最畅销的IT书籍作者之一Thomas Erl联合云计算专家和研究者，详细分析了业已证明的、成熟的云计算技术和实践，并将其组织成一系列定义准确的概念、模型、技术机制和技术架构，所有这些都是以工业为中心但是与厂商无关的。

本书理论与实践并重，重点放在主流云计算平台和解决方案的结构和基础上。除了以技术为中心的内容以外，还包括以商业为中心的模型和标准，以便读者对基于云的IT资源进行经济评估，把它们与传统企业内部的IT资源进行比较。此外，本书提供了一些用来计算与SLA相关的服务质量的模板和公式，还给出了大量的SaaS、PaaS和IaaS交付模型。

本书包括超过260幅图、29个架构模型和20种机制，是一本不可或缺的指导书，是对云计算技术的详细解读。

云计算与分布式系统：从并行处理到物联网

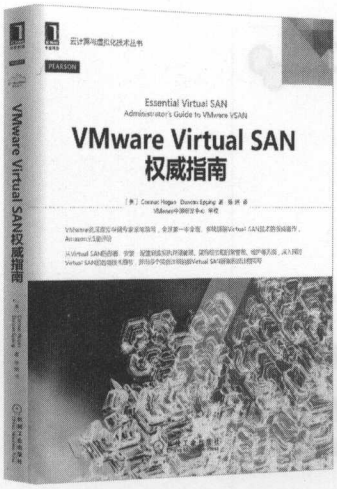
作者：Kai Hwang 等 译者：武永卫 等 ISBN：978-7-111-41065-2 定价：85.00元

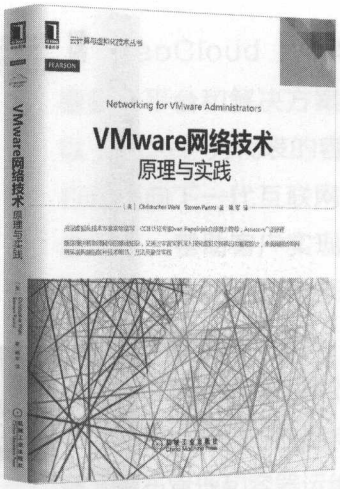
“本书是一本全面而新颖的教材，内容覆盖高性能计算、分布式与云计算、虚拟化和网格计算。作者将应用与技术趋势相结合，揭示了计算的未来发展。无论是对在校学生还是经验丰富的实践者，本书都是一本优秀的读物。”

—— Thomas J. Hacker，普度大学

本书是一本完整讲述云计算与分布式系统基本理论及其应用的教材。书中从现代分布式模型概述开始，介绍了并行、分布式与云计算系统的设计原理、系统体系结构和创新应用，并通过开源应用和商业应用例子，阐述了如何为科研、电子商务、社会网络和超级计算等创建高性能、可扩展的、可靠的系统。

推荐阅读





VMware Virtual SAN权威指南

作者：（美）Cormac Hogan 等 ISBN：978-7-111-48023-5 定价：59.00元

不论您是虚拟化新手，还是存储专家，这本书是有关VMware Virtual SAN最权威的解读，是实现软件定义存储最有效的指南。

——任道远，VMware中国研发中心总经理

VMware资深虚拟存储专家亲笔撰写，全球第一本全面、系统讲解Virtual SAN技术的权威著作，Amazon全5星评价。

从Virtual SAN的部署、安装、配置到虚拟机存储管理、架构细节和日常管理、维护等方面，深入探讨Virtual SAN的各项技术细节，并用多个实例详细讲解Virtual SAN群集的设计和实现。

本书专为管理员、咨询师和架构师所著，在书中Cormac Hogan和Duncan Epping既介绍了Virtual SAN如何实现基于对象的存储和策略平台，这些功能简化了虚拟机存储的放置，还介绍了Virtual SAN如何与vSphere协同工作，大幅提高系统弹性、存储横向扩展和QoS控制的能力。

VMware网络技术：原理与实践

作者：（美）Christopher Wahl 等 ISBN：978-7-111-47987-1 定价：59.00元

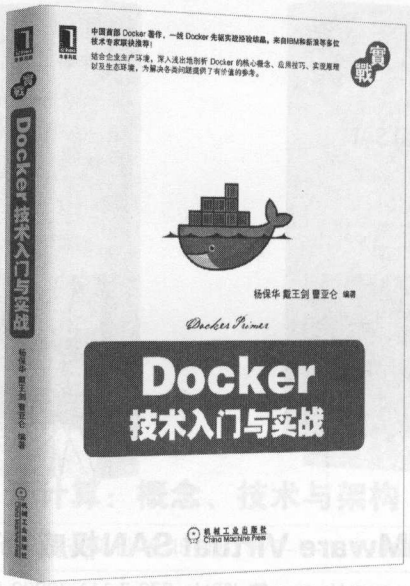
资深虚拟化技术专家亲笔撰写，CCIE认证专家Ivan Pepelnjak作序鼎力推荐，Amazon广泛好评

既详细讲解物理网络的基础知识，又通过丰富实例深入探究虚拟交换机的功能和设计，

全面阐释虚拟网络环境构建的各种技术细节、方法及最佳实践

本书针对VMware专业人员，阐述了现代网络的核心概念，并介绍了如何在虚拟网络环境设计、配置和故障检修中应用这些概念。作者凭借其在虚拟化项目实施方面的丰富经验，从网络模型、常见网络层次的介绍开始，由浅入深地介绍了现代网络的基本概念，并自然地过渡到虚拟交换等虚拟化环境中与物理网络最为关联的部分，最后扩展到实际的设计用例，详细介绍了不同实用场景、不同的硬件配置下，虚拟化环境构建的考虑因素和具体实施方案。

推荐阅读



Docker 技术入门与实战

作者：杨保华 等 书号：978-7-111-48852-1 定价：79.00元

中国首部Docker著作，一线Docker先驱实战经验结晶，
来自IBM和新浪等多位技术专家联袂推荐！
结合企业生产环境，深入浅出地剖析 Docker 的核心概念、应用技巧、
实现原理以及生态环境，为解决各类问题提供了有价值的参考。

本书作者之一杨保华博士在加入 IBM之后，一直从事云计算与软件定义网络领域的相关解决方案和核心技术的研发，热心关注 OpenStack、Docker 等开源社区，热衷使用开源技术，并积极参与开源社区的讨论、积极提交代码。这使得他既能从宏观上准确把握 Docker 技术在整个云计算产业中的定位，又能从微观上清晰理解技术人员所渴望获知的核心之处。

—— 刘天成，IBM中国研究院云计算运维技术研究组经理

好的IT技术总是迅速“火爆”，Docker就是这样。好像忽然之间，在企业一线工作的毕业生们都在谈论Docker。在IT云化的今天，系统的规模和复杂性，呼唤着标准化的构件和自动化的管理，Docker正是这种强烈需求的产物之一。这本书很及时，相信会成为IT工程师的宝典。

—— 李军，清华大学信息技术研究院院长

DaoCloud 是业界领先的企业级容器云平台和解决方案提供商，致力于以 Docker 为代表的容器技术，为企业打造面向下一代互联网应用的交付和运维平台，帮助客户实现云端持续创新。DaoCloud 采用混合云模式，以云端 SaaS 化容器管理平台，对接各类主机资源，构建跨云跨网的容器主机资源池，提供全流程标准化的应用持续集成、镜像构建、发布管理和容器运维服务。欢迎访问 https://www.daocloud.io了解更多。

微信公众号

DaoCloud

封面设计 · 杨宇梅

本书通过分析解读Docker源码，让读者了解 Docker的内部结构和实现，以便更好地使用Docker。该书的内容组织深入浅出，表述准确到位，有大量流程图和代码片段帮助读者理解Docker各个功能模块的流程，是学习Docker开源系统的良师益友。

—— **寿黎旦** 浙江大学计算机学院教授

近年来，Docker 迅速风靡了云计算世界，但是专门针对 Docker 的技术实现进行深入分析的文章则相对较少。……本书恰如其时的出现，弥补了这个空白，对于希望参与到 Docker 社区、参与代码贡献或构建自己的 Docker 应用环境的读者来说，应是一本案头必备书籍。

—— **王兴宇** 《Linux中国》创始人

在崇尚代码至上的工程师文化里，文档介绍、发布会材料都是苍白的，唯有研读代码，才能深刻理解软件背后的原理。与所有其他软件一样，读代码并不是学习Docker最快的途径，但是如果有人通读代码后给出了详细分析，你就可以轻松地站在巨人的肩膀上。

很高兴看到国内这么快就出版了代码分析的书籍。对于所有想在Docker方面进阶和想晋升为高端使用者的用户，都值得阅读本书。也希望通过这本书，可以诞生更多的社区贡献者，共同推动Docker的发展。

—— **黄 强** 华为Docker Committer

这本书从源码的角度对Docker的实现原理进行了深入的探讨和细腻的讲解，将当前炙手可热的容器技术的背后机理讲解得深入浅出、明白透彻。无论是Docker的使用者还是开发者，通过阅读此书都可以对Docker有更深刻的理解，能够更好地使用或者开发Docker。

—— **雷继棠** 华为Docker Committer

Docker已经是一个成长两年的云计算技术，它正在以惊人的速度在全球范围内扩张自己的“疆土”。我作为Docker中国区的开发者，非常希望能看到一本书详细地告诉我，Docker的每一个细节是如何实现的。……我希望你能在这本书上学到更多Docker技术的精髓思想，在实战Docker技术时可以运用自如！

—— **肖德时** 数人科技CTO

我家里的书柜中至今仍然保留着一本《Linux内核完全注释》……，它使我受益匪浅。10年后，当我拿到宏亮的《Docker源码分析》草稿，昨日又重现。本书无论是对学习Docker，还是Go语言，都是非常好的一手资源。雷锋不常有，大家要支持！

—— **赵 鹏** VisualOps创始人

上架指导：计算机/云计算
ISBN 978-7-111-51072-7
9 787111 510727 >
定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259
华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn